Technical Document 2533
May 1993

Unix STREAMS Emulation of an Input/Output Controller (IOC) for an Embedded AN/UYK-44(V) Processor

D. R. Wilcox
P. N. Pham

Technical Document 2533
May 1993

Unix STREAMS Emulation of an Input/Output Controller (IOC) for an Embedded AN/UYK-44(V) Processor

D. R. Wilcox
P. N. Pham

NAVAL COMMAND, CONTROL AND OCEAN SURVEILLANCE CENTER
RDT&E DIVISION
San Diego, California 92152–5001

K. E. EVANS, CAPT, USN
Commanding Officer

R. T. SHEARER
Executive Director

ADMINISTRATIVE INFORMATION

Work for this report was performed by members of the Computer Systems Branch Code 412 within the Information Processing and Displaying Division Code 41 of NCCOSC RDT&E Division, San Diego, California, during the period of October 1992 through May 1993. The work was funded under Project Element: OMN 0305167G by the Naval Space and Warfare Systems Command, Code 398277, Washington DC 20360–5100.

Released by
D. G. Sheriff, Head
Computer Systems Branch

Under authority of
A. G. Justice, Head
Information Processing
and Displaying Division

ACKNOWLEDGEMENTS

The authors wish to thank Joel Melohn and Dan Stout of Paramax for providing independent testing of the software presented in this technical document; Viet Tran of NRaD Code 412 for detecting and documenting the VMEbus read–modify–write problem in the DTC–2 computer; and David Sheriff of NRaD Code 412 for his technical and textual review of the final draft of this technical document.

RV

CONTENTS

FIGURES

1.0 INTRODUCTION

This technical document presents the implementation of a software interface between a Unix operating system that is executing on a Navy DTC−2 VMEbus computer workstation and a VMEbus AN/UYK−44(V) enhanced processor (EP) embedded within that workstation.

Figure 1 shows the hardware configuration. All the hardware shown reside within the DTC−2 chassis. The VME AN/UYK−44(V) EP and VME random access memory (RAM) boards use the standard 6U form factor and are located in the upper card cage. The Sun−4 processor board uses the 9U form factor and is located in the lower card cage. The Sun−4 board employs the SPARC microprocessor. The Sun−4 board provides the VMEbus system controller function. The upper card cage is at the far end of the VMEbus arbitration and interrupt daisy chains.

The VME AN/UYK−44(V) EP board does not have any local memory. It executes programs from the VME RAM board. The VME RAM board contains two megabytes of memory. It is configured with a VMEbus base byte address of 10 000 000 hexadecimal. The AN/UYK−44(V) EP does have a cache to help minimize VMEbus traffic. The VME Sun−4 board has both a local memory and a cache. It is programmed to access the VME RAM board when it needs to share data with the VME AN/UYK−44(V) EP board.

The software interface between the Sun−4 and AN/UYK−44(V) EP applications uses the Unix STREAMS mechanism to emulate the AN/UYK−44(V) input/output controller (IOC) function within the Sun−4. The AN/UYK−44(V) application directs the interface by supply-

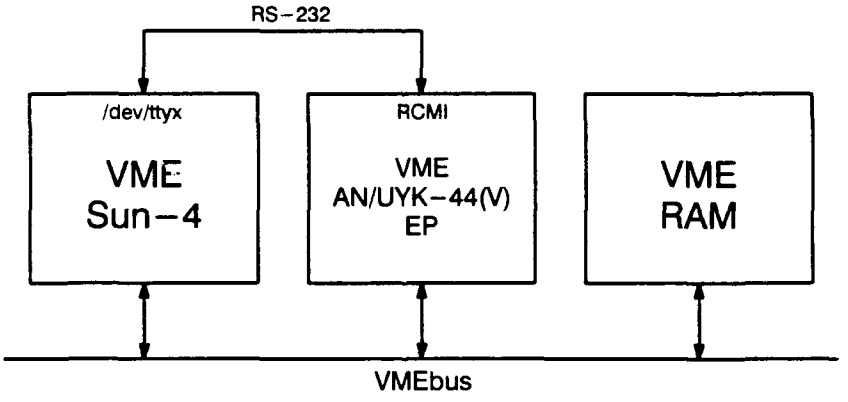


Figure 1. AN/UYK−44(V) embedment within a DTC−2 VMEbus computer workstation

ing AN/UYK-44(V) command instructions and chain programs within the VME RAM. The Unix application communicates with the interface by using the Unix file system calls.

2.0 APPLICATION INTERFACES

This section presents the Unix application interface and the AN/UYK−44(V) application interface for the Unix STREAMS emulation of the AN/UYK−44(V) input/output controller.

2.1 READ OUTPUT AND WRITE INPUT DATA TRANSFERS

The AN/UYK−44(V) IOC channels appear to the Unix application as sequential files. Each IOC channel file has an entry in the `/dev` directory. The emulation supports four IOC's, with 16 channels per IOC, giving a total of 64 `/dev` entries. The file names are of the form "`iocx.y`", where *x* is the AN/UYK−44(V) IOC number, and *y* is the hexadecimal channel number on IOC *x*. Channel 15 on IOC 2, for example, has file name "`ioc2.f`".

The Unix application establishes and releases access to the AN/UYK−44(V) channel files by using the Unix `open` and `close` system calls, respectively. The Unix application sends data to the AN/UYK−44(V) by using the Unix `write` system call, and obtains data from the AN/UYK−44(V) by using the Unix `read` system call.

The following program reads output data from AN/UYK−44(V) IOC 2 channel 15 and displays it in hexadecimal format.

```
#include <fcntl.h>

#define FILENAME "/dev/ioc2.f"
#define BUFSIZE 16

main()
{
  int fd;
  char buffer[BUFSIZE];
  int i, n;

  if((fd = open(FILENAME, O_RDWR)) < 0) {
    perror("open failed");
    exit(1);
  }

  if((n = read(fd, buffer, sizeof(buffer))) < 0) {
    perror("read failed");
    exit(2);
  }

  for(i = 0; i < n; i++) {
    (void)printf(" %02x", buffer[i] & 0xFF);
```

```
    }
    (void)putchar('\n');
}
```

2.2 EXTERNAL FUNCTION DATA TRANSFERS

The AN/UYK-44(V) IOC distinguishes between normal transfers and external function transfers. When there is a requirement to read AN/UYK-44(V) external function transfers, the Unix `getmsg` system call must be used instead of the Unix `read` system call. The `getmsg` system call separates the "control part" from the "data part" of a stream message by placing the parts into different buffers. For the AN/UYK-44(V) IOC emulation, the stream message associated with a normal transfer has a data part but no control part, while that associated with an external function transfer has a control part but no data part. The Unix application can distinguish between normal and external function reads by determining which stream message part the `getmsg` system call receives.

The AN/UYK-44(V) IOC also distinguishes between external functions and forced external functions. The emulation processes forced external functions as "high-priority" stream messages. This means that forced external function messages will be received before other types of messages when messages are queued waiting to be accepted by the Unix application.

The following program reads, identifies, and displays output data, external function data, and forced external function data from AN/UYK-44(V) IOC 0 channel 3.

```
#include <fcntl.h>
#include <stropts.h>

#define FILENAME "/dev/ioc0.3"
#define RDBUFSIZE 16
#define EFBUFSIZE 2

main()
{
  int fd;
  struct strbuf rdbuf;
  struct strbuf efbuf;
  char rddata[RDBUFSIZE];
  char efdata[EFBUFSIZE];
  int flags;
  int i;

  if((fd = open(FILENAME, O_RDWR)) < 0) {
    perror("open failed");
    exit(1);
  }
```

```
    rdbuf.maxlen = sizeof(rddata);
    rdbuf.buf = rddata;

    efbuf.maxlen = sizeof(efdata);
    efbuf.buf = efdata;

    flags = 0;

    if(getmsg(fd, &efbuf, &rdbuf, &flags) < 0) {
      perror("getmsg failed");
      exit(2);
    }

    if(rdbuf.len > 0) {
      (void)printf("UYK-44 output data\n");

      for(i = 0; i < rdbuf.len; i++) {
        (void)printf(" %02x", rddata[i] & 0xFF);
      }
      (void)putchar('\n');
    }

    if(efbuf.len > 0) {
      if(flags == RS_HIPRI) {
        (void)printf("UYK-44 force external function data\n");
      }
      else {
        (void)printf("UYK-44 external function data\n");
      }

      for(i = 0; i < efbuf.len; i++) {
        (void)printf(" %02x", efdata[i] & 0xFF);
      }
      (void)putchar('\n');
    }
  }
```

2.3 EXTERNAL INTERRUPT GENERATION

The Unix application sends an external interrupt to the AN/UYK-44(V) through a channel by using the `ioctl` system call with the `I_STR` parameter. The call takes a pointer to a `strioctl` structure as an argument. The include file `<stropts.h>` defines the `strioctl` structure. Structure member `ic_dp` is a pointer to the data, and structure member `ic_len` is the size of the data associated with the call. For the external interrupt, the data is the value of the external interrupt word stored by the external interrupt into AN/UYK-44(V) memory. The size of the data is used to determine whether a 16-bit word or a 32-bit double-word is stored in the AN/UYK-44(V) memory.

The AN/UYK-44(V) IOC swaps the two 16-bit halves of 32-bit external interrupt words before storing them. There is no such thing as an 8-bit external interrupt word. Since 8-bit transfers use the 8 least-significant data bits on 16-bit parallel channel interface hardware, it is logical to use the 8 least-significant bits of the 16-bit external interrupt words when emulating 8-bit peripherals. This decision, however, is left to the programmer.

The following program sends a 16-bit external interrupt to AN/UYK-44(V) IOC 0 channel 3. The external interrupt word has a value of `0x1234`.

```
#include <local/epcntl.h>

#include <fcntl.h>
#include <stropts.h>

#define FILENAME "/dev/ioc0.3"
#define EIWORD 0x1234

main()
{
  int fd;
  struct strioctl cntl;
  unsigned short eiword;

  if((fd = open(FILENAME, O_RDWR)) < 0) {
    perror("open failed");
    exit(1);
  }

  eiword = EIWORD;

  cntl.ic_cmd = EXT_INTR;
  cntl.ic_timout = 5;
  cntl.ic_dp = (char *)&eiword;
  cntl.ic_len = sizeof(eiword);

  if(ioctl(fd, I_STR, &cntl) < 0) {
    perror("ioctl failed");
    exit(2);
  }
}
```

2.4 IOC COMMAND INSTRUCTIONS

The VME AN/UYK-44(V) EP initiates execution of an IOC command instruction emulation by first loading a command block at a dedicated location in the VME RAM, and then by sending a VMEbus priority level 2 interrupt to Unix to indicate that the command block is ready for processing. Figure 2 shows the format of the IOC emulation command block.

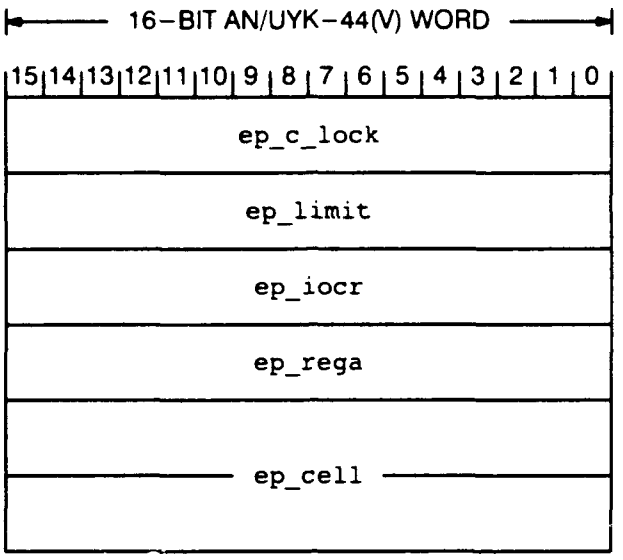
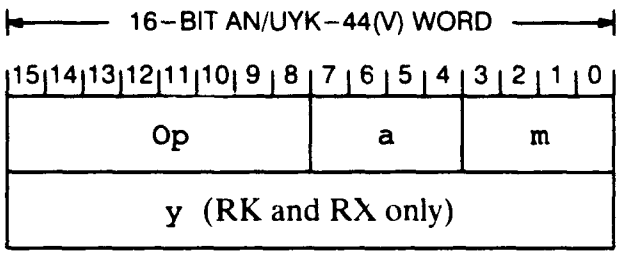


Figure 2. IOC emulation command block format

Since the command block is a resource shared between the AN/UYK-44(V) and Unix, coordination is required to insure that the AN/UYK-44(V) does not enter a new command before Unix has captured the content of the previous command. Before entering a new command, the AN/UYK-44(V) should test the `ep_c_lock` word using the AN/UYK-44(V) processor biased fetch instruction. The command block is available when the biased fetch instruction reads bits 14 and 15 as zero. After performing its read, the biased fetch instruction writes ones into bits 14 and 15 to mark the command block as busy. The emulation clears the `ep_c_lock` word, including bits 14 and 15, after its interrupt service routine has captured the command block contents.

The `ep_limit` word specifies the maximum number of emulation cycles permitted before the chain is automatically terminated. Emulation cycles are explained later in section 3.5. The `ep_limit` word is intended as a debugging aid. The `ep_limit` word is ignored when set to zero.

The command block `ep_iocr` and `ep_rega` words are patterned after the operation of the AN/UYK-44(V) processor `IOCR` and `IOC` instructions. Bits 3 through 0 of the `ep_iocr` word indicate the channel number. When bits 7 through 4 are zero, the IOC number is zero and the `ep_rega` word is ignored. When bits 7 through 4 are not zero, the IOC number is read from bits 1 through 0 of the `ep_rega` word.



Op a m	Mnemonic	Format	Description
E0 - 0	ACR 0	RR	Clear All Channels
E0 - 4	ACR 4	RR	Enable EI Data Transfers on All Channels
E0 - 5	ACR 5	RR	Disable EI Data Transfers on All Channels
E0 - 6	CCR a,6	RR	Enable Lower Channel Interrupts
E0 - 7	CCR a,7	RR	Disable Lower Channel Interrupts
E0 - 8	CCR a,8	RR	Clear Channel a
E0 - 9	CCR a,9	RR	Clear Input on Channel a
E0 - A	CCR a,A	RR	Clear Output on Channel a
E0 - C	CCR a,C	RR	Enable EI Data Transfer on Channel a
E0 - D	CCR a,D	RR	Disable EI Data Transfer on Channel a
E0 - E	CCR a,E	RR	Enable Interrupts on Channel a
E0 - F	CCR a,F	RR	Disable Interrupts on Channel a
E6 - -	WIMK a,y,m	RK	Write Channel Control Register
E6 - 2	ICK a,y	RK	Initiate Input Chain
E6 - 6	OCK a,y	RK	Initiate Output Chain
E7 - -	WIM a,y,m	RX	Write Channel Control Register
EB - -	RIM a,y,m	RX	Read Channel Control Register
FB - -	SST a,y,m	RX	Store Channel Status Register

Figure 3. Emulated AN/UYK-44(V) IOC command instruction formats

The two `ep_cell` words correspond to the AN/UYK-44(V) IOC command cell. They hold the command instruction to be emulated. Figure 3 shows the format and the list of implemented IOC command instructions.

The initiate input chain and the initiate output chain command instructions, `ICK` and `OCK`, start the execution of input and output IOC chain program emulations, respectively. The number of input and output chain programs executing simultaneously is limited only by the number of configured channels, which is typically 64.

The emulation does not execute an AN/UYK-44(V) IOC command when the file associated with the AN/UYK-44(V) channel on which the command is to execute has not been

opened by the Unix application. Requests to execute a command associated with an unopened channel file are discarded.

2.5 IOC CHAIN PROGRAMS

The AN/UYK–44(V) IOC emulation executes chain programs located in the AN/UYK–44(V) memory. The chain programs are coded using a subset of the AN/UYK–44(V) IOC instruction set. Figure 4 shows the format and lists the AN/UYK–44(V) IOC instructions that are implemented by the emulation.

The emulation is incorporated within the Unix kernel. Since the emulation blindly follows the chain program, care must be exercised when coding chain programs containing infinite loops.

> **NOTE:** Executing a chain program containing an infinite loop will cause the Unix kernel to enter an infinite loop.

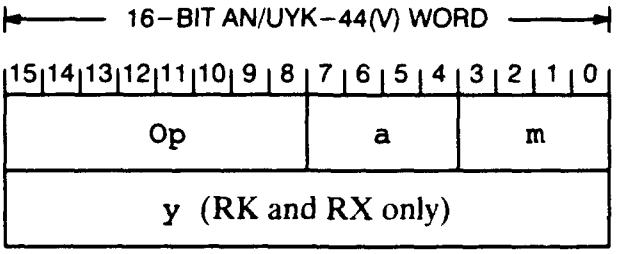
Not all infinite loops are bad. The ones that cause trouble are those without some form of flow control. A chain program consisting of an unconditional branch instruction branching to itself will cause the kernel to spin indefinitely. The only way to escape such a loop is to send a clear channel command from the AN/UYK–44(V), or to hardware master clear the Unix system. On the other hand, loops containing data transfer instructions are less likely to cause trouble because they interact with stream queues. Flow control on these queues can suspend the chain when the queue becomes full or empty.

2.6 AN/UYK–44(V) INTERRUPT GENERATION

There are a number of instances where the AN/UYK–44(V) IOC emulation sends an interrupt to the AN/UYK–44(V) EP hardware. These include external interrupts initiated by the Unix application, and input chain, output chain, instruction fault, and memory protection fault interrupts initiated by the emulation.

A board on the VMEbus generates an interrupt within the VME AN/UYK–44(V) EP board by writing into any of the 64 mailbox locations recognized by the VME AN/UYK–44(V) EP board. The AN/UYK–44(V) IOC emulation uses one of these mailbox locations. The VME AN/UYK–44(V) EP board recognizes the mailbox address. The VME RAM board stores the mailbox data.

The VME AN/UYK–44(V) EP requires all 64 mailbox locations to be the same size. The size may be 2, 4, 8, or 16 bytes. The SPARC processor, however, can only perform 1–byte, 2–byte, and 4–byte VMEbus accesses. The emulation performs a 4–byte write into the mail-



Op	a	m	Mnemonic		Format	Description
E0	–	0	ACR	0	RR	Clear All Channels
E0	–	4	ACR	4	RR	Enable EI Data Transfers on All Channels
E0	–	5	ACR	5	RR	Disable EI Data Transfers on All Channels
E0	–	6	CCR	a,6	RR	Enable Lower Channel Interrupts
E0	–	7	CCR	a,7	RR	Disable Lower Channel Interrupts
E0	–	8	CCR	a,8	RR	Clear Channel a
E0	–	9	CCR	a,9	RR	Clear Input on Channel a
E0	–	A	CCR	a,A	RR	Clear Output on Channel a
E0	–	C	CCR	a,C	RR	Enable EI Data Transfer on Channel a
E0	–	D	CCR	a,D	RR	Disable EI Data Transfer on Channel a
E0	–	E	CCR	a,E	RR	Enable Interrupts on Channel a
E0	–	F	CCR	a,F	RR	Disable Interrupts on Channel a
E3	0	–	IO	0,y	RX	Initiate Input Transfer
E3	1	–	IO	1,y	RX	Initiate Output Transfer
E3	2	–	IO	2,y	RX	Initiate External Function Transfer
E3	3	–	IO	3,y	RX	Initiate Force External Function Transfer
E6	–	–	LCMK	m,y	RK	Load Channel Control Register
E7	–	–	LCM	m,y	RX	Load Channel Control Register
EB	–	–	SCM	m,y	RX	Store Channel Control Register
EC	0	–	HCR		RR	Halt Chain
EC	1	–	IPR		RR	Interrupt Processor
EF	0	–	ZF	y	RX	Zero Flag
EF	1	–	SF	y	RX	Set Flag
EF	2	–	TF	y	RX	Test Flag
EF	4	–	ZB	y,m	RX	Zero Bit
EF	5	–	SB	y,m	RX	Set Bit
EF	7	–	CB	y,m	RX	Compare Bit
F2	0	–	SJC	0,y	RK	Unconditional Jump
F2	4	–	SJMC	4,y	RK	Jump on Status Condition Set
F2	8	–	SJMC	8,y	RK	Jump on Active Input Transfer
F2	9	–	SJMC	9,y	RK	Jump on Active Output Transfer
FB	–	–	CSST	y,m	RX	Store Channel Status Register

Figure 4. Emulated AN/UYK-44(V) IOC chain instruction formats

box dedicated to it. Thus, any mailbox size equal to or larger than 4 bytes can be used. A mailbox size larger than 4 bytes may be required by some other board in the VMEbus system not presented here.

The emulation passes information that identifies the interrupt to the AN/UYK−44(V) through the interrupt parameter block. Figure 5 shows the format of the interrupt parameter block. The emulation tests the `ep_i_lock` word to see if it is set to zero. If the `ep_i_lock` word is not zero, the emulation waits on the assumption that the AN/UYK−44(V) is still processing the previous interrupt. If the `ep_i_lock` word is zero, the emulation loads the other words of the interrupt parameter block, then sets all the bits of the `ep_i_lock` word to ones, and finally writes the AN/UYK−44(V) absolute 16−bit word address of the interrupt parameter block into the mailbox to trigger the AN/UYK−44(V) interrupt. It is the responsibility of the software executing on the AN/UYK−44(V) to clear the `ep_i_lock` word once it has captured the contents of the interrupt parameter block.

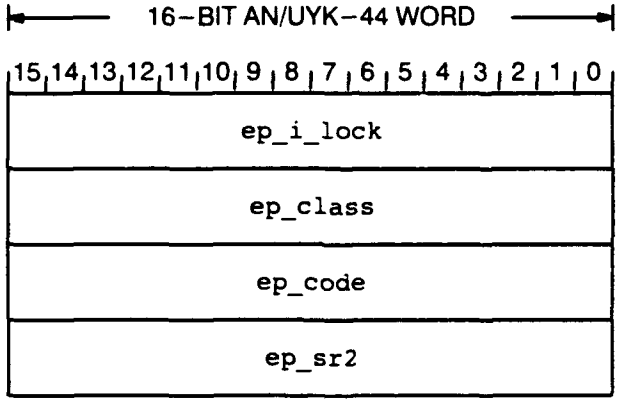


Figure 5. IOC interrupt parameter block format

2.7 READ−MODIFY−WRITE INSTRUCTION RESTRICTIONS

The zero flag (`ZF`), set flag (`SF`), test flag (`TF`), zero bit (`ZB`), set bit (`SB`), and compare bit (`CB`) chain instructions are implemented on the AN/UYK−44(V) IOC using 16−bit read−modify−write memory cycles. Unfortunately, some processor instruction sets, including that of the SPARC used in the DTC−2, are not capable of emulating these instructions exactly. The only available SPARC instructions employing read−modify−write memory cycles are the `ldstub`, `ldstuba`, `swap`, and `swapa` instructions.

The SPARC `ldstub` and `ldstuba` instructions atomicly read the contents from a memory byte location into a register and write ones back into all the bits at the same memory byte loca-

tion. The byte contents are treated as an unsigned number. Although the details differ, the `ldstub` and `ldstuba` instructions are intended to be used in the same manner as the AN/UYK-44(V) processor biased fetch instructions. The six read-modify-write AN/UYK-44(V) IOC chain instructions involve both setting and clearing individual bits, and not merely setting an entire byte. The `ldstub` and `ldstuba` instructions are not much help.

The SPARC `swap` and `swapa` instructions atomicly exchange the contents of a register with a 32-bit memory word. Although exact emulation of the six read-modify-write AN/UYK-44(V) IOC chain instructions on the SPARC is not possible, one can obtain atomic coordination between programs executing on the two architectures by placing restrictions on how the `swap` and `swapa` instructions are used. The two rules are as follows:

(1) There can be no more than one flag within any aligned 32-bit word. A flag here means a two-state indicator, which both architectures are capable of interpreting, regardless of the actual number of bits involved.

(2) All the remaining non-flag bits in the aligned 32-bit word must be either constant or not used.

The first rule requires dedicating a full 32-bit word because that is the only data aggregate size that the SPARC `swap` instruction can manipulate. To understand the reason for the second rule, consider how the `swap` instruction works. The `swap` instruction atomicly exchanges the contents of the 32-bit flag word with the contents of a register. The register contents become the flag word contents. But the flag and non-flag portions of this 32-bit transfer cannot be separated. The `swap` instruction moves all 32 bits together. The problem is in knowing what the non-flag bits should be in the register before executing the `swap` instruction. If the non-flag bits are either not used or are constant, there is no problem. But if they are variable, they must be read beforehand. Unfortunately, they may change between the time that they are written into the register and the time that the register contents are transferred back to the flag word by the `swap` instruction. The process is not atomic. The only way to make it atomic is with another `swap` instruction, which leads us back to where we started. The second rule prevents the problem by requiring that the non-flag bits be either not used or constant.

Although the Sun-4 processor board within the DTC-2 uses a SPARC microprocessor, it does not implement the `swap` instruction using an atomic VMEbus read-modify-write cycle. The VMEbus read-modify-write cycle requires that the AS* signal remain low between the read and write portions of the cycle to prevent other processors from gaining access. Monitoring the VMEbus signals with a logic analyzer reveals that the AS* signal goes high between the read and write portions of the cycle.

As a consequence of the failure of the Sun-4 board to implement the VMEbus read-modify-write cycle, the DTC-2 emulation of the zero flag (`ZF`), set flag (`SF`), test flag (`TF`), zero

bit (ZB), set bit (SB), and compare bit (CB) AN/UYK-44(V) IOC chain instructions are not atomic.

2.8 CHANNEL TYPES

In the AN/UYK-44(V) IOC architecture, channel status word 0 bits 7 through 4 specify the channel type. On the hardware version of the AN/UYK-44(V) IOC, channel types are determined by the particular IOA hardware module plugged into the AN/UYK-44(V) backplane. On the emulation presented here, the Unix application can set the channel type by using an `ioctl` system call with the `CHANNEL_TYPE` parameter. The `CHANNEL_TYPE` parameter is defined in the include file `"epcntl.h"`. The argument is a byte containing the channel type in the four least-significant bits. As always, the `ioctl` system call takes the address of the argument, not the argument itself. The emulation remembers the channel type even after the Unix file for the channel is closed.

3.0 IOC EMULATION ARCHITECTURE

This section presents the structure and concepts employed in the emulation of the AN/UYK−44(V) IOC within the framework of the STREAMS facility of the Unix operating system.

The presentation below assumes a basic understanding of the operation of the Unix operating system and STREAMS facility. Publications explaining Unix are widely available. The bibliography at the end of this document lists a number of textbooks for readers unfamiliar with the construction of stream drivers and modules.

3.1 EMULATION STREAM MODULES

Figure 6 shows the interconnection of the stream modules used for the AN/UYK−44(V) IOC emulation.

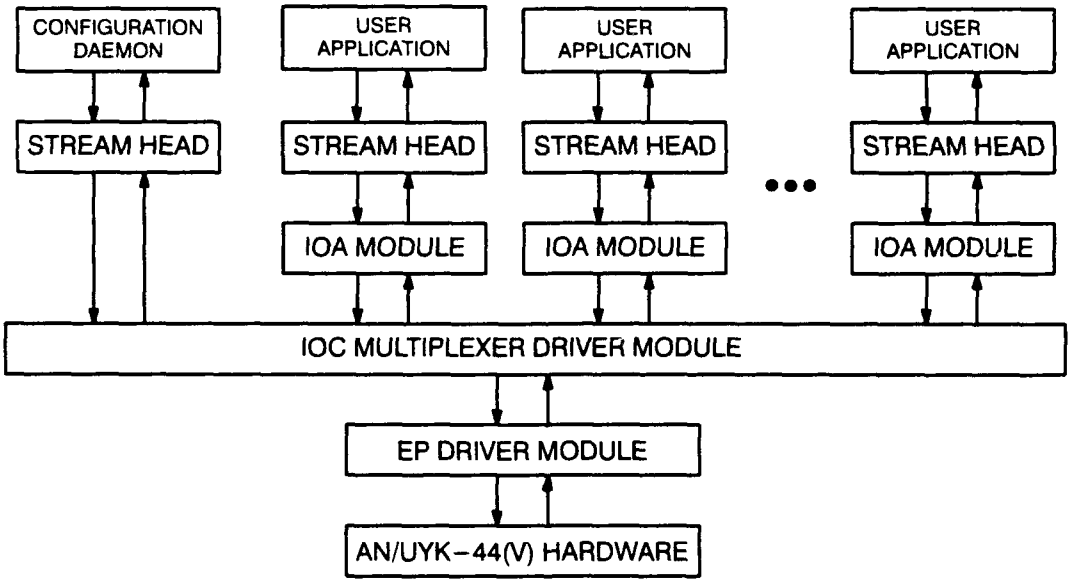


Figure 6. STREAMS interface to the AN/UYK−44(V) EP

The EP driver module provides the interface between the Unix STREAMS facility and the AN/UYK−44(V) Enhanced Processor (EP) hardware. The EP driver module contains the emulation of the AN/UYK−44(V) IOC page registers. It implements page register address mapping and access error detection for the emulated AN/UYK−44(V) IOC memory accesses

into the VME RAM board. It converts the command block interrupt received over the VMEbus from the AN/UYK−44(V) EP into a stream message for use by upstream modules. It triggers interrupts within the AN/UYK−44(V) EP in response to stream messages assigned for that purpose from upstream modules. It also triggers interrupts within the AN/UYK−44(V) EP for memory access faults detected during page register memory address mapping.

The IOC multiplexer driver module directs stream message traffic between the many upstream IOA modules and the single downstream EP driver module. IOC multiplexer driver module port 0 is reserved for the configuration daemon program. The configuration daemon program maintains the connection between the IOC multiplexer driver module and the EP driver module independently of the open status of the other ports. The IOC multiplexer driver module contains the emulation of the channel control and status registers.

The IOA modules, one for the emulation of each respective AN/UYK−44(V) IOC channel, interpret the command, input chain, and output chain instructions. They are automatically pushed into the stream when an IOC multiplexer driver module port is opened.

3.2 FILE NAMES AND MINOR DEVICES

The file names associated with the IOC multiplexer driver module ports reflect the AN/UYK−44(V) IOC and channel numbers. File `/dev/ioc2.3`, for example, is associated with AN/UYK−44(V) IOC 2 channel 3. A Unix application program that reads AN/UYK−44(V) output data or writes AN/UYK−44(V) input data on AN/UYK−44(V) IOC 2 channel 3 opens file `/dev/ioc2.3`. Channel numbers larger than 9 use lower case hexadecimal equivalents. IOC 2 channel 15, for example, is assigned file `/dev/ioc2.f`.

Two files that are not intended to be accessible by the application are `/dev/ioc0`, which is reserved for communication between the IOC multiplexer driver module and the configuration daemon program, and `/dev/ep0`, which is reserved for communication between the EP driver module and the IOC multiplexer driver module. The EP driver module and the IOC multiplexer driver module permit only one open at a time to these respective files.

Each IOC multiplexer driver module port has a unique minor device number. Minor device 0 is reserved for use by the configuration daemon program. The other minor device numbers are assigned to emulations of the various AN/UYK−44(V) IOC channels. To convert an AN/UYK−44(V) IOC number and channel number into a minor device number, multiply the IOC number by 16, then add the channel number, and finally add one. IOC 2 channel 3, for example, has minor device number $2 \times 16 + 3 + 1 = 36$ decimal, or 24 hexadecimal.

3.3 IOC PAGE REGISTERS

All AN/UYK-44(V) addresses point to 16-bit words rather than to bytes. The AN/UYK-44(V) IOC maps 16-bit AN/UYK-44(V) relative addresses into 22-bit AN/UYK-44(V) absolute addresses by using page registers. Each page register points to a 1K-word page of memory. Pages are aligned on 1K-word boundaries. The page register additionally specifies the read, write, and execute restrictions on access to the page through that page register and provides a bit which is set to one whenever the page is modified. Figure 7 shows the AN/UYK-44(V) IOC page register format.

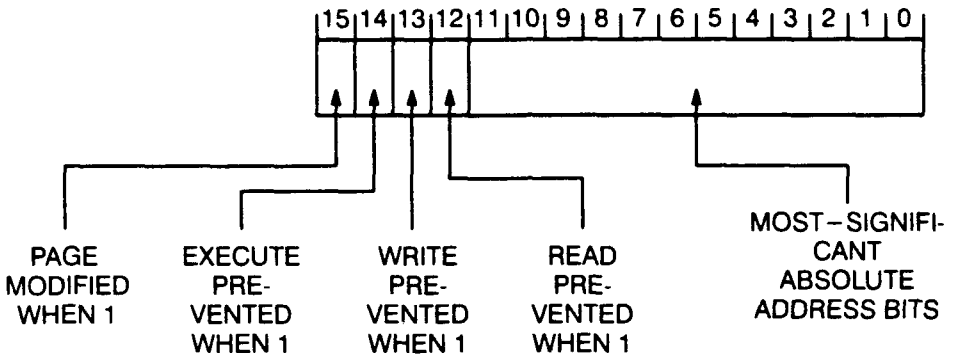


Figure 7. AN/UYK-44(V) IOC page register format

The AN/UYK-44(V) 16-bit relative address defines a 64K-word relative address space. This 64K-word space is fully mapped by 64 page registers. Such a group of 64 page registers is called a page register group. The AN/UYK-44(V) implements four IOC page register groups. The set selected depends on the IOC instruction and channel number involved.

The AN/UYK-44(V) IOC relative-to-absolute address mapping process consists of using the 6 most-significant bits of the relative address to select one of the 64 page registers in the page register group, and then concatenating the 12 least-significant bits of the selected page register to the 10 least-significant bits of the relative address to form the 22-bit absolute address. Figure 8 illustrates the process.

The EP driver module emulates the four IOC page register groups defined by the AN/UYK-44(V) instruction set architecture. The page register groups are mapped into the highest addresses of the VME RAM board. The AN/UYK-44(V) EP can read or write the IOC page registers simply by reading or writing at the respective memory locations. The EP driver module reads these locations when it converts AN/UYK-44(V) 16-bit relative addresses into AN/UYK-44(V) 22-bit absolute addresses. The EP driver module writes into these page register locations to set the most-significant bit, known as the page modification bit, when it writes into a memory page pointed to by the respective page register.

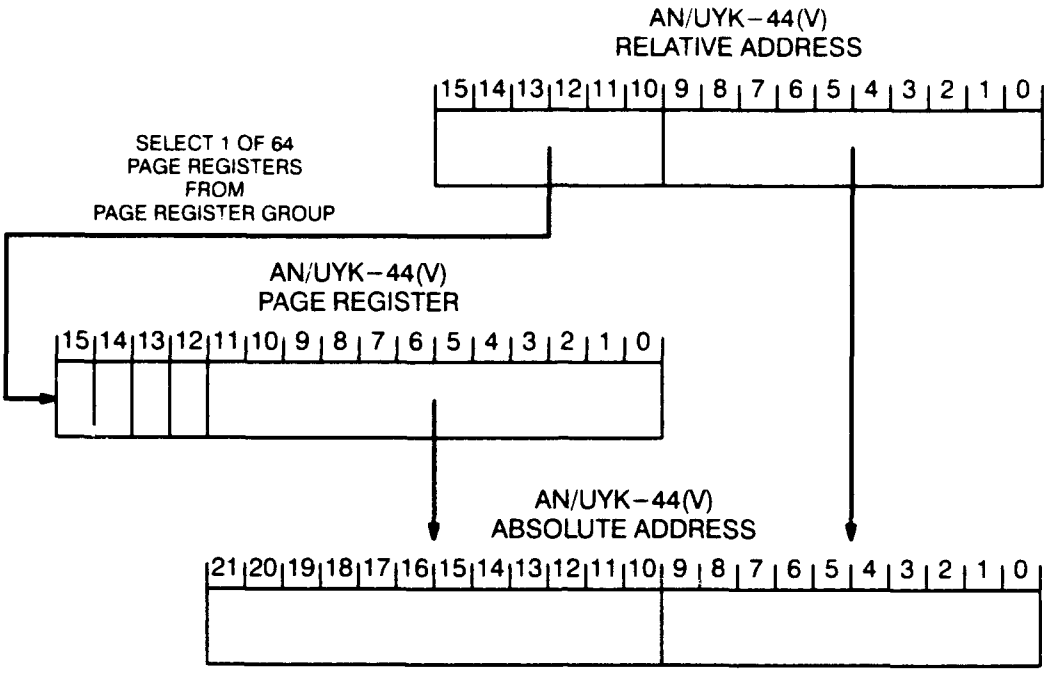


Figure 8. AN/UYK-44(V) relative-to-absolute address mapping

The EP driver sets the page modification bit by reading the location dedicated to the page register, setting the most-significant bit, and then storing the resulting page register contents back into the same memory location. Although it may seem otherwise, this read-modify-write operation does not need to be atomic. This is because it makes no sense for an AN/UYK-44(V) application to be both using a page register and asynchronously modifying it. When the page register is in use, only writes that set the page modification bit should occur. When the page register is not in use, only writes that modify the page register should occur. The writes in the two cases should be mutually exclusive. There should never be a write between the read and write portions of the operation setting the page modification bit. It is not essential, therefore, to provide read-modify-write atomicity at the VMEbus level.

Although the 22-bit AN/UYK-44(V) absolute address defines a 4M-word address space, the VME RAM board employed provides only 1M words. The space available to the application is further limited, albeit only slightly, by the dedication of 256 words to the emulation of the four IOC page register groups described above, and by the dedication of a few words to buffers for communication of IOC emulation information between Unix and the AN/UYK-44(V) EP. The EP driver module, when properly configured, prevents access by IOC command instructions and chain programs to non-existent and dedicated memory locations.

Within the EP driver module, the VME RAM board appears as a one-dimensional array of C language `unsigned short` elements. The driver accesses the VME RAM board by using the AN/UYK-44(V) absolute address as the array index. The driver accesses the array in the same manner that other Unix drivers access hardware control and status registers. The VMEbus hardware base byte address of the VME RAM board is `0x10000000`. Figure 9 illustrates the various levels of address translation from the AN/UYK-44(V) 16-bit relative word address to the VMEbus 32-bit hardware byte address.

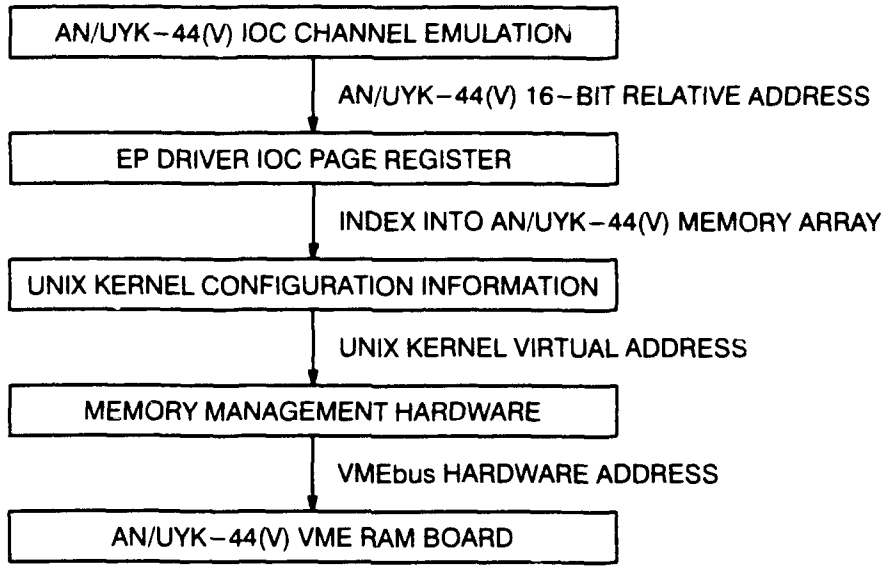


Figure 9. AN/UYK-44(V) to VMEbus address translation

3.4 CHANNEL CONTROL REGISTERS

The IOC multiplexer driver module contains the kernel memory dedicated to the channel control registers. There is one set of 16 channel control registers for each channel. The channel control registers were not located in the EP driver module because it has no knowledge of the number of channels actually configured. The channel control registers were not placed in the IOA modules because there are some channel command and chaining instructions that affect all channels, or all channels with a priority lower than a specified channel.

Channel control registers 0 contains the buffer control word for transfers from Unix to the AN/UYK-44(V). Transfers in this direction are "input" transfers from the AN/UYK-44(V) viewpoint and "write" transfers from the Unix viewpoint.

Figure 10 shows the format of the buffer control word. The transfer mode field indicates the size of each unit of transfer between an AN/UYK−44(V) IOC and a peripheral device. It determines the number of bits transferred by each cycle of the hardware interface signals of the channel being emulated. The transfer count field indicates the number of such cycles in the block transfer. The emulation moves data between memories, rather than between memory and a physical peripheral device. The emulation does not give significance to the transfer mode field contents other than to scale the transfer count field contents during its computation of the total size of the block being transferred.

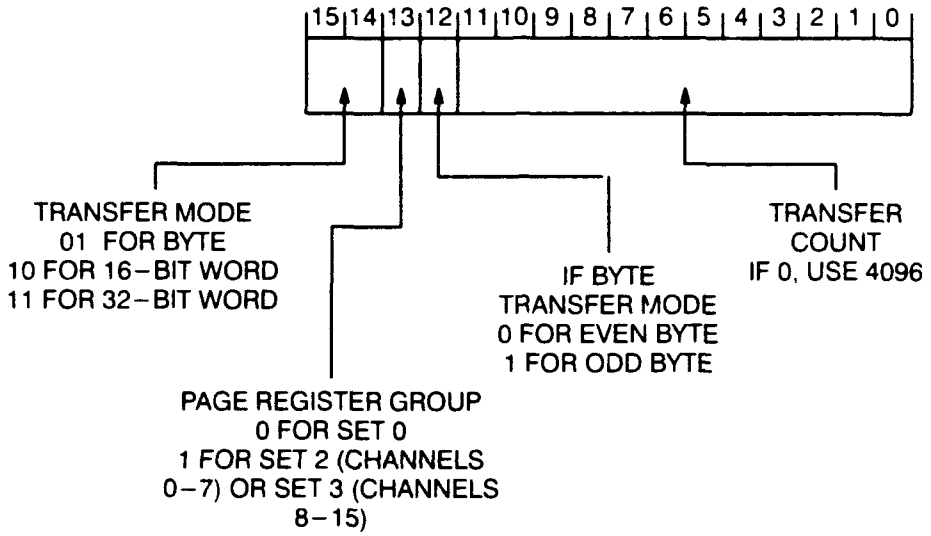


Figure 10. Buffer control word format

Channel control register 1 contains the buffer address pointer for the transfer. It indicates the 16−bit AN/UYK−44(V) relative starting address where the data obtained from Unix is to be placed.

Channel control registers 4 and 5 contain the buffer control word and buffer address pointer, respectively, for data transfers from the AN/UYK−44(V) to Unix. These are "output" transfers from the AN/UYK−44(V) viewpoint, and "read" transfers from the Unix viewpoint.

Conceptually, the transfer count decrements and the buffer address pointer increments as a transfer takes place. The IOC emulation, however, only updates their values at the beginning and end of the transfer, or, if the transfer prematurely terminates, when it terminates. This is done for efficiency so that the EP driver module does not need to report each individual byte, word, or double−word transfer to the IOC multiplexer driver module, but merely the results of the entire data block transfer.

Channel control registers 2 and 6 contain the chain address pointers for the input and output chains, respectively. A chain address pointer is a program counter for a chain program. It points to the 16-bit AN/UYK-44(V) relative address of the next chain instruction in the chain program.

The other channel control registers are implemented, but are not assigned a specific function by the IOC multiplexer driver module.

3.5 EMULATION MESSAGES

Each chain program execution is emulated by circulating an independent stream message in a loop between the IOA module and either the EP driver or the IOC multiplexer driver module. The EP driver module creates the message when it processes a command block interrupt from the AN/UYK-44(V) EP. The IOC multiplexer driver module destroys the message when the chain executes a halt or clear instruction or detects a fatal error.

Figure 11 shows the emulation message format. Most emulation messages have only an `M_PROTO` block. Forced external function messages use an `M_PCPROTO` block instead of an `M_PROTO` block for higher priority within stream queues. Messages transporting input, output, or external function application data append one, or more, `M_DATA` blocks to the `M_PROTO` or `M_PCPROTO` block. `M_DATA` blocks are variable in length.

The format of the `M_PROTO` or `M_PCPROTO` block remains the same throughout the life of the chain program execution. Nearly all messages that the IOA module receives from the IOC multiplexer driver module are simply modified and sent back downstream. The common format allows use of the same memory allocation for both the received and the sent message. Even if a few bytes of memory are sometimes never used, this is more efficient than deallocating memory from the received message, and then allocating memory for a new message. The common message format also avoids the need to copy fields that are common to both messages from the received message to the sent message.

The message `function` field indicates the operation performed by the message as it passes through the various stream modules. Each time the message arrives at the IOA module, it is assigned the next function to perform. As the message circulates from the IOA module downstream and then back upstream to the IOA module, the modules along the way perform their portion of the function execution. The only case when the message `function` field is not set by the IOA module is when a new message is created.

The message `ack` field indicates whether the message should continue to circulate or be discarded after its function has been executed. This field enables the module completing the function execution to discard immediately those messages no longer needed, rather than send-

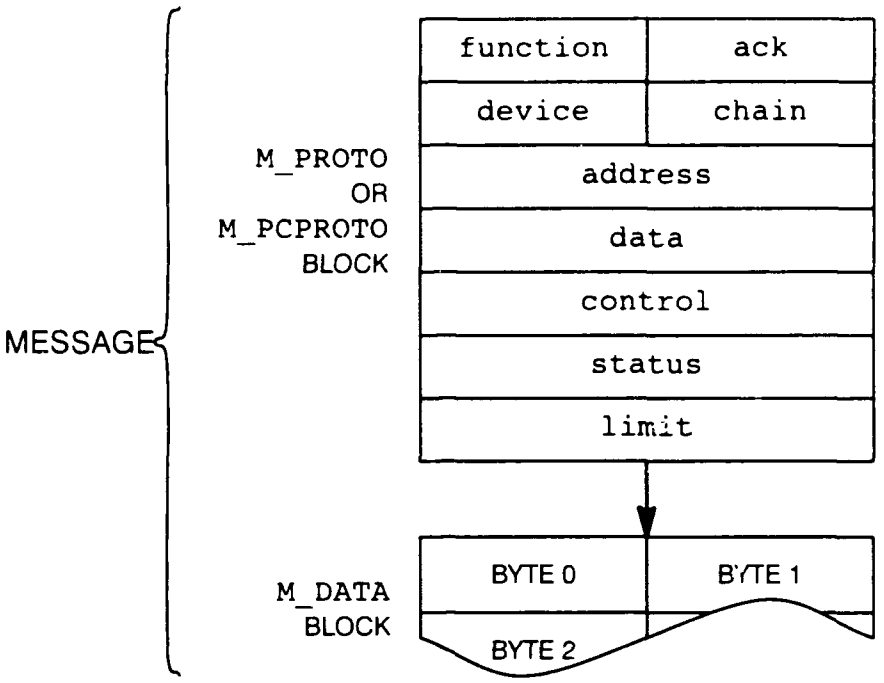


Figure 11. Emulation stream message format

ing them back upstream to the IOA module, only to have them discarded there. The message `ack` field has a value of either `REPLY` or `DISCARD`. In general, chain program emulations set the field to `REPLY`, and command instruction emulations set the field to `DISCARD`.

The message `device` field indicates the Unix minor device number of the AN/UYK−44(V) IOC channel file. The IOC multiplexer driver module sets the message `device` field as the message passes downstream based on the queue from which it obtained the message. The IOC multiplexer driver module reads the `device` field as the message passes upstream to determine the queue to which the message should be returned.

The message `chain` field indicates whether the message is associated with an input chain or an output chain. It has a value of either `INPUT` or `OUTPUT`.

The message `address`, `data`, and `control` fields provide the operands of the operation specified by the message `function` field. Although given the titles "address", "data", and "control", the actual use of these fields is less restrictive than their titles may imply. Figure 12 and Figure 13 summarize the field assignments.

The message `status` field reports the memory access faults detected by the EP driver module. These faults include memory protection faults, based on the contents of the respective

function	address	data	control
`IO_CELL` IO command cell	second command cell word	not used	first command cell word
`READ_INST` read next instruction	instruction address	instruction contents	not used
`READ_INST_Y` read instruction y field	y field address	y field content	first instruction word contentss
`JUMP_INST` jump and read next instruction	instruction address	instruction contents	not used
`JUMP_STAT` jump if status bit set and read next instruction	instruction address	instruction contents	status register mask
`READ_REG` read control register	not used	register contents	register number
`WRITE_REG` write control register	not used	write data	register number
`REG_TO_MEM` copy control register to memory	memory address	register contents	register number
`MEM_TO_REG` copy memory to control register	memory address	memory contents	register number
`WRITE_STAT` write status register	status regieter number	write data	write mask
`STAT_TO_MEM` copy status register to memory	memory address	status register contents	status register number

Figure 12. STREAMS emulation message field assignments

function	address	data	control
READ_START read buffer control word and buffer address pointer	buffer address pointer	address of buffer control word	buffer control word
READ_BUF read memory buffer	buffer address pointer	not used	buffer control word
WRITE_START write buffer control word and buffer address pointer	buffer address pointer	address of buffer control word	buffer control word
WRITE_BUF write memory buffer	buffer address pointer	not used	buffer control word
EF_START read external function buffer control word and buffer address pointer	buffer address pointer	address of buffer control word	buffer control word
EF_BUF read external function buffer	buffer address pointer	not used	buffer control word
WRITE_WORD write memory word through mask	memory address	memory data	write mask
TEST_WORD test memory word through mask	memory address	memory read data	memory data mask
TEST_AND_SET test and set memory word through mask	memory address	memory write data	memory data mask
INTERRUPT generate AN/UYK−44(V) interrupt	interrupt code	interrupt class	CP status register 2
CHAN_CONTROL channel control	channel select bits	control bits set	control bit clear mask
WRITE_EI_WORD write external interrupt word	32−bit EI word extension	EI word data	size of EI word in bytes

Figure 13. STREAMS emulation message field assignments (cont'd)

page register memory protection bits, and memory range faults, caused by attempts to access non-supported AN/UYK-44(V) memory addresses.

The message `limit` field indicates the number of message circulations remaining before the message is automatically discarded. This message circulation count is a debugging feature that can be disabled by setting the `limit` field to zero.

3.6 CHAIN PROGRAM EMULATION EXAMPLE

Emulating a chain program execution by using a circulating stream message can be illustrated with a specific example. Consider an output chain program that performs an output data block transfer of `BUFSZ` 16-bit words, starting at AN/UYK-44(V) memory address `BUF`, that interrupts the AN/UYK-44(V) when the transfer is completed, and finally that halts. The chain program, in AN/UYK-44(V) assembly language, is a follows:

```
CHAIN    IO       1,BCWBAP . initiate output transfer
         IPR               . interrupt EP
         HLT               . halt chain

BCWBAP   +08000X+BUFSZ     . for output buffer control word
         +BUF              . for output buffer address pointer

         EVEN
BUF      RES      BUFSZ    . EP memory buffer
```

The AN/UYK-44(V) EP hardware interrupts the EP driver module to inform it of the initiate output chain command awaiting execution. The EP driver module creates a new stream message to accept the information provided with the AN/UYK-44(V) EP interrupt. The AN/UYK-44(V) EP provides the IOC number and the location of the command cell. The EP driver module uses the IOC number obtained directly and the channel number obtained by accessing the command cell to compute the minor device number. It loads the minor device number into the message `device` field. The minor device number established by the EP driver module remains with the message throughout the life of the chain program execution. The message `device` field enables the IOC multiplexer driver module to direct the message into the proper IOA module when the message passes upstream. The EP driver module also loads the contents of the command cell into the appropriate message fields so that they can be decoded and processed by the upstream IOA module. Finally, the EP driver module sets the `function` field to the constant `IOCELL` to identify the message contents as those for a command cell message. The fields of the newly created message are as follows:

```
function IOCELL
ack      REPLY
device   computed by EP driver module
```

chain	not used
address	second 16-bit word of command cell
data	not used
control	first 16-bit word of command cell
status	not used
limit	0

When the IOCELL message reaches the IOA module, the IOA module decodes the first 16-bit word of the command cell carried by the message to determine the AN/UYK-44(V) command instruction desired. In this case, the AN/UYK-44(V) command is an initiate output chain instruction. The IOA module sets the ack field to REPLY and the chain field to OUTPUT. The second 16-bit word of the command cell gives the address where the output chain program is to begin execution. This address must be sent to the output chain address pointer channel control register, which acts as the program counter for the chain program. The IOA module sets the function field to WRITE_REG to request the transfer. The fields of the message sent downstream by the IOA module are now as follows:

function	WRITE_REG
ack	REPLY
device	no change
chain	OUTPUT
address	not used
data	new contents for output chain address pointer
control	OUT_CAP (channel control register 6)
status	not used
limit	0

The channel control registers are implemented within the IOC multiplexer driver module. When the WRITE_REG message reaches the IOC multiplexer driver module, the IOC multiplexer driver module recognizes that the function involves only one of its own registers. It extracts the register number, in this case OUT_CAP, and the data to be loaded into that register from the appropriate message fields. Since the message ack field is set to REPLY, and since no further processing is necessary downstream, the IOC multiplexer driver module returns the message upstream to the IOA module.

Upon receiving the returned WRITE_REG message, the IOA module is now ready to request access of the first instruction of the chain program. This is accomplished by setting the function field to READ_INST. The IOA module does not need to modify any of the other fields. The fields of the message are now as follows:

function	READ_INST
ack	still set to REPLY
device	no change
chain	still set to OUTPUT
address	not used

data	not used
control	not used
status	not used
limit	0

When the READ_INST message arrives at the IOC multiplexer driver module, the IOC multiplexer driver module sets the message address field to the contents of its output chain address pointer channel control register. It then increments the contents of the chain address pointer channel control register in anticipation of the next chain instruction request. Finally, it sends the message downstream to the EP driver module. The fields of the message are as follows:

function	READ_INST
ack	still set to REPLY
device	no change
chain	still set to OUTPUT
address	output chain address pointer from IOC multiplexer driver module
data	not used
control	not used
status	not used
limit	0

The EP driver module maps the 16-bit address specified by the message address field into a location within the AN/UYK-44(V) memory. The AN/UYK-44(V) instruction set architecture defines all chain program instructions as executing through IOC page register group 0. The mapping through the page register by the EP driver module also checks for execute protect fault and for the page register pointing to non-existent locations in AN/UYK-44(V) memory. The EP driver module stores the results of these tests in the message status field. Assuming no faults, the EP driver module loads the contents of the memory location into the message data field and then returns the message to the IOC multiplexer driver module.

function	READ_INST
ack	still set to REPLY
device	no change
chain	still set to OUTPUT
address	output chain address pointer from IOC multiplexer driver module
data	first chain instruction word read by EP driver module
control	not used
status	EP_NO_FAULTS
limit	0

The IOC multiplexer driver module, in turn, returns the message to the IOA module selected by the message device field without further processing.

The IOA module now processes the first instruction of the output chain program. It decodes the instruction opcode and discovers that it defines an initiate output transfer instruc-

tion. Since an initiate output transfer instruction is a two-word instruction, and since so far the IOA module has only the first 16-bit word, its next message circulation needs to obtain the second 16-bit word. The IOA module moves the first word of the chain instruction from the message `data` field to the message `control` field for safe keeping. This word transfer frees the message `data` field to capture the second word of the chain instruction. The IOA module also changes the message `function` field to `READ_INST_Y`. When the message eventually returns to the IOA module, the new message `function` field informs the IOA module that the message now contains both the first and the second words of the two-word chain instruction. When the recirculated message finally returns to the IOA module, its fields contain the following:

`function`	`READ_INST_Y`
`ack`	still set to `REPLY`
`device`	no change
`chain`	still set to `OUTPUT`
`address`	output chain address pointer from IOC multiplexer driver module
`data`	second chain instruction word read by EP driver module
`control`	first chain instruction word saved by IOA module
`status`	`EP_NO_FAULTS` for second chain instruction word
`limit`	`0`

The IOA module again decodes the opcode and discovers that the chain instruction is an initiate output transfer instruction. Since it has the entire chain instruction this time, it uses the next message circulation to execute the instruction.

Before describing the message that executes the instruction, it is helpful to review the operation performed by the initiate output transfer instruction in the AN/UYK-44(V) IOC. The second 16-bit word of the initiate output transfer instruction points to two consecutive 16-bit words in memory. These two words contain the values to be loaded by the instruction into the output buffer control word and the output buffer address pointer channel control registers, respectively. In the example, they are located at address `BCWBAP`. After loading them, the instruction starts the block data transfer they define. The buffer control word gives the size of the transfer and the buffer address pointer gives the AN/UYK-44(V) memory starting address of the data block transferred.

The execution portion of the initiate output transfer instruction emulation requires two message circulations.

For the first message circulation, the IOA module sets the message `function` field to `READ_START`. Remember that from the stream viewpoint, the stream *reads* AN/UYK-44(V) output data. As stated, the second 16-bit word of the initiate output transfer instruction points to two consecutive 16-bit words containing the values to be loaded into the output buffer control word and output buffer address pointer channel control registers, respectively. The ad-

dress of these words already exists in the message `data` field from the previous message circulation. The `START_READ` message passes downstream through the IOC multiplexer driver module to the EP driver module. At the EP driver module, the contents of the two words at the address specified by the message `data` field are accessed and inserted into the message. As the message passes upstream and back through the IOC multiplexer driver module, the IOC multiplexer driver module copies the initial values for the output buffer control word and the output buffer address pointer from the message into the respective channel control registers within the IOC multiplexer driver module. The IOC multiplexer driver module also sets a bit in the channel status register to indicate that the channel is active. Finally, the message returns to the IOA module with the following contents:

`function`	`READ_START`
`ack`	still set to `REPLY`
`device`	no change
`chain`	still set to `OUTPUT`
`address`	value for buffer address pointer read by EP driver module
`data`	`BCWBAP` pointer from IOA module
`control`	value for buffer control word read by EP driver module
`status`	`EP_NO_FAULT` on both EP driver module reads
`limit`	`0`

For the second message circulation, the IOA module sets the message `function` field to `READ_BUF`. When the message reaches the EP driver module, the EP driver module allocates a stream `M_DATA` block and loads it with data in accordance with the specification provided by the output buffer control word and the output buffer address pointer contained in the message fields. The EP driver module appends the loaded `M_DATA` block to the `M_PROTO` block of the message. The EP driver module also updates the contents of the message fields containing the output buffer control word and the output buffer address pointer to reflect the extent that the transfer was completed. Finally, it sends the message back upstream. At the IOC multiplexer driver module, the contents of the output buffer control word and the output buffer address pointer channel control registers are updated using the values provided by the message fields, and the channel status register output channel active flag is cleared. The message fields are as follows:

`function`	`READ_BUF`
`ack`	still set to `REPLY`
`device`	no change
`chain`	still set to `OUTPUT`
`address`	new value of buffer address pointer from EP driver module
`data`	not used
`control`	new value of buffer control word from EP driver module
`status`	`EP_NO_FAULTS` on any EP driver module reads
`limit`	`0`

When the IOA module receives the returning READ_BUF message, it splits the message into two messages by detaching the M_DATA block from the M_PROTO block. The IOA module changes the function field of the message formed from the M_PROTO block to READ_INST and then circulates it to obtain the next chain instruction. It sends the message formed from the M_DATA block upstream to the stream head where the contents become the data accepted by the Unix read system call.

The next chain instruction in the example, IPR, interrupts the AN/UYK−44(V) EP processor. Its emulation requires two message circulations. The first message, with the function field set to READ_INST, reads the IPR instruction itself. The second message, with the function field set to INTERRUPT, directs the EP driver module to interrupt the AN/UYK−44(V) EP. The EP driver module triggers the interrupt by writing into a VMEbus mailbox recognized by the AN/UYK−44(V) EP hardware. An additional message, with the function field set to READ_INST_Y, is not necessary, in this case, because the IPR instruction is a one−word instruction.

Each channel has the ability to enable or disable its input chain interrupt, output chain interrupt, and external interrupt by executing a command or chain instruction. If channel interrupts are enabled, the INTERRUPT message from the IOA module passes without delay through the IOC multiplexer driver module to the EP driver module. If channel interrupts are disabled, however, the INTERRUPT message is not sent to the EP driver module, but rather sets a pending interrupt flag in the IOC multiplexer driver module. When a message to enable interrupts eventually appears at the IOC multiplexer driver module, the pending interrupt flags are examined. If any pending interrupt flags indicate pending interrupts, respective INTERRUPT messages are generated and sent downstream to the EP driver module.

The final chain instruction in the example, HLT, halts the chain program. As with all chain instruction emulations, its first message circulation is with the function field set to READ_INST to obtain the identity of the instruction itself. Upon identifying the instruction as a HLT instruction, the IOA module sets the function field to CHAN_CONTROL and the ack field to DISCARD. When the message reaches the IOC multiplexer driver module, it sets a flag to indicate that the chain is halted. Since the message ack field is set to DISCARD, the message is not sent back to the IOA module. Terminating message circulation terminates the chain program.

3.7 AN/UYK−44(V) MEMORY ACCESS ERRORS

There are four possible AN/UYK−44(V) memory access errors. They are read protect fault, write protect fault, execute protect fault, and address range fault. The EP driver module detects these errors when it processes messages requiring access to the AN/UYK−44(V)

memory. The EP driver module sends a memory protect interrupt to the AN/UYK-44(V) EP when it detects a read, write, or execute protect fault. It sends a memory resume interrupt to the AN/UYK-44(V) EP when it detects a memory range fault.

When the EP driver module detects an AN/UYK-44(V) memory access error during message processing, it sets the message `status` field to indicate the presence and nature of the error. The message propagates upstream to the IOA module. When the IOA module sees a memory access error reported in the message `status` field, it converts the the stream message type from `M_PROTO` to `M_ERROR`. The first byte of the `M_ERROR` message is set to the error code `EIO` defined by the Unix `read` and `write` system calls. Finally, the IOA module sends the message upstream to the stream head. Since the message is not sent back downstream, it no longer circulates, which effectively terminates the chain.

3.8 FLOW CONTROL

The Unix `read` and `write` system calls obtain the Unix application buffer size from a call parameter. The AN/UYK-44(V) IOC initiate transfer chain instructions obtain the AN/UYK-44(V) application buffer size from the buffer control word addressed by the instruction. There is no requirement that the application buffer sizes match. The STREAMS system implements read and write queues. Data can be enqueued in blocks of one size and dequeued in blocks of another size.

As shown by a previous example, AN/UYK-44(V) output data made available for Unix reading is simply attached to the `READ_BUF` message by the EP driver module and is subsequently removed and sent to the stream head by the IOA module. The stream head automatically handles any mismatch between the size of the message data block received from downstream and the size of the application buffer specified within the Unix `read` system call.

Unix write data made available for AN/UYK-44(V) input is attached to a `WRITE_BUF` message by the IOA module and is subsequently removed and stored into AN/UYK-44(V) memory by the EP driver module. Several factors, all related to flow control, make attaching write data blocks to the `WRITE_BUF` message considerably more complex, in comparison to attaching read data blocks to the `READ_BUF` message.

The first problem concerns queuing the Unix write data. The input chain does not accept data until it executes an initiate transfer chain instruction. If the Unix write data arrives at the IOA module before the execution of this instruction, the Unix write data must be held in a queue.

The IOA module write queue receives both the Unix data messages from the stream head and the circulating `WRITE_BUF` messages from the IOA module read service routine. The

Unix data messages have stream message type M_DATA. The circulating WRITE_BUF messages are temporarily given the stream message type M_PCPROTO. The high-priority M_PCPROTO message type is used, rather than the normal-priority M_PROTO message type, to insure that the circulating message will get into the IOA module write queue when the queue is in the flow control blocking state. This is important to avoid deadlock. Otherwise, the queue may be filled with Unix data messages before the WRITE_BUF message arrives, preventing the WRITE_BUF message from entering for lack of space. The M_PCPROTO message type overrides the flow control high water mark. It also places the WRITE_BUF message at the front of the queue. This makes searching for it faster. The WRITE_BUF message type is set back to M_PROTO when it leaves the queue.

The data messages within the IOA module write queue are in first-in-first-out order. The WRITE_BUF message, if present, is at the front of the queue. The stream **getq** routine returns whatever message is at the front of the queue. It could return either a WRITE_BUF message, if a WRITE_BUF message is in the queue, or a data message, if no WRITE_BUF message is in the queue. But the only messages that should be released downstream are WRITE_BUF messages with attached data blocks. Data messages should be held in the queue until an appropriate WRITE_BUF message becomes available to which they can be attached. The IOA module write queue, therefore, cannot be treated simply as a first-in-first-out queue using the stream **getq** routine.

The problem does not exist when going in the other direction for Unix read data. Unix read data is always available in the AN/UYK-44(V) memory. It is accessed from the AN/UYK-44(V) memory when the initiate transfer chain instruction is executed. Any necessary queuing of the Unix read data, while awaiting the Unix **read** system call to accept it, takes place at the stream head.

The second problem concerns the holding of the WRITE_BUF and data messages in the write queue. A Unix write data transfer to the AN/UYK-44(V) memory cannot take place until there is *both* sufficient Unix write data and a AN/UYK-44(V) IOC initiate transfer chain instruction to accept the write data. As already shown, if its write data arrives before the instruction, the write data is queued until the instruction arrives. Similarly, if the instruction arrives before there is sufficient write data, the instruction is queued until *all* of its required write data arrives. The IOA module write service routine must deal with both possibilities.

When a data message arrives from the stream head, the IOA module write service routine checks the front of the write queue to see if there is a WRITE_BUF message in the queue. If there is, the IOA module write service routine sequentially adds the sizes of any data messages in the write queue to see if there is now sufficient write data available to satisfy the WRITE_BUF message buffer size requirements. If this is also the case, the IOA module write service routine appends the appropriate write data blocks to the WRITE_BUF message and sends it down-

stream. If any of these conditions are not met, messages are neither removed from the write queue nor sent downstream.

Similarly, when a WRITE_BUF message arrives from the IOA module read service routine, the IOA module write service routine examines the data messages in the write queue to see if there is already sufficient write data available to satisfy the WRITE_BUF message buffer size requirements. If there is, the IOA module write service routine appends the appropriate write data blocks to the WRITE_BUF message and sends it downstream. If not, messages are neither removed from the write queue nor sent downstream.

The third problem concerns the possible mismatch between the size of the buffer specified through the Unix `write` system call and that specified by the buffer control word addressed by the initiate transfer chain instruction. The stream head delivers to the IOA module M_DATA messages that are the same size as the buffer size parameter passed by the application to the Unix `write` system call that created them. The IOA module write service routine concatenates and splits M_DATA messages, as required, to fulfill the needs requested by the WRITE_BUF message.

3.9 CHANNEL CLEAR CHAIN TERMINATION

The AN/UYK-44(V) IOC instruction set includes command and chain instructions to clear all the channels, a particular channel, the input portion of a particular channel, and the output portion of a particular channel. Since a channel clear chain instruction executing on one channel can clear another channel, channel clearing is implemented in the IOC multiplexer driver module where all channels are accessible.

The IOC multiplexer driver module has four flags for each channel. These flags are named IN_CH_ACTIVE, IN_CH_ABORT, OUT_CH_ACTIVE, and OUT_CH_ABORT.

The IN_CH_ACTIVE flag indicates whether or not a channel input chain is currently executing on the channel. The OUT_CH_ACTIVE flag indicates whether or not a channel output chain is currently executing on the channel. They are set by the respective initiate chain instructions and are cleared by halt chain instructions in the respective chains.

The IN_CH_ABORT flag indicates whether or not the channel has received a request from a channel clear instruction to terminate the currently executing input chain. The OUT_CH_ABORT flag indicates whether or not the channel has received a request from a channel clear instruction to terminate the currently executing output chain. They are set by clear channel instructions directed to the channel, regardless of the channel or command source of the instruction.

Chain program execution continues as long as the stream message emulating its execution continues to circulate. Destroying the circulating message terminates the chain. Each time a

downstream emulation message enters the IOC multiplexer driver module, the IOC multiplexer driver module examines either the two flags associated with input chains or the two associated with output chains. The message `chain` field determines the pair of flags examined.

If the flags simultaneously indicate that there is a chain currently executing on the channel and that a request to clear the channel has been received, the IOC multiplexer driver module destroys the message to terminate the chain. It clears the two flags to indicate that the channel no longer has an active chain and that the channel clear has been completed.

If, on the other hand, the flags simultaneously indicate that there is not a chain currently executing on the channel and that a request to clear the channel has been received, the IOC multiplexer driver module does not destroy the message since the message is initiating a new chain or performing a command. It sets the active chain flag if the message is initiating a new chain. It clears the abort flag to indicate that the channel clear has been completed, or more precisely, was never necessary in the first place because the channel was already cleared.

Finally, if the abort flag indicates that a request to clear the channel has not been received, then channel clearing is not required.

3.10 EXTERNAL INTERRUPT

The Unix application generates an external interrupt for the AN/UYK−44(V) by using the Unix `ioctl` system call. The external interrupt word, associated with the external interrupt, is passed as the parameter to the `ioctl` system call. The stream head converts the `ioctl` system call into an `M_IOCTL` message with the external interrupt word appended in a separate `M_DATA` block.

The IOA module processes the `M_IOCTL` message. It first checks the size of the external interrupt word. If the `M_DATA` block, containing the external interrupt word, is not one, two, or four bytes long, the IOA module assumes it to be bad, deletes the `M_DATA` block, converts the `M_IOCTL` block into an `M_IOCNAK` block, and returns the `M_IOCNAK` block as a message to the stream head. Upon receiving the `M_IOCNAK` message, the stream head returns an error to the application.

On the other hand, if the size of the `M_DATA` block containing the external interrupt word is correct, the IOA module generates two new `M_PROTO` message blocks. The first has `WRITE_EI_WORD` in its `function` field. It contains the value and size of the external interrupt word. The second `M_PROTO` message block has `INTERRUPT` in its `function` field. It is similar to the message used for the `IPR` interrupt described previously. The second message block is appended to the first, so that the two message blocks can be sent downstream as a single message. The IOA module sets the block `ack` fields to `DISCARD`, since they are not part

of a chain that must circulate a message. The IOA module does not need to specify the block `device` field, because they are automatically loaded by the IOC multiplexer driver module downstream. Finally, the IOA module converts the `M_IOCTL` message into an `M_IOCACK` message and returns it to the stream head.

At the IOC multiplexer driver module, the two blocks forming the message are processed separately. The `INTERRUPT` block is processed in the same manner as other interrupts. If interrupts are enabled, the block is sent on its way to the EP driver module. If interrupts are not enabled, a flag is set to indicate that the interrupt is pending and the block is discarded. The `WRITE_EI_WORD` block is processed in the same manner. If the flag indicating the state of the EIE line is enabled, the block is sent on its way to the EP driver module. If the EIE line is disabled, the value of the external interrupt word is stored in the IOC multiplexer driver module, a flag indicating a pending external interrupt word transfer is set, and the `WRITE_EI_WORD` block is discarded. If both blocks are being sent downstream to the EP driver module, the `INTERRUPT` block is appended to the `WRITE_EI_WORD` block, so that they can be sent as a single message.

4.0 INSTALLATION AND INVOCATION

This section presents code used to install and invoke the STREAMS emulation of the AN/UYK−44(V) IOC. It is assumed that the reader is familiar with Unix, shell commands, and the C programming language.

4.1 CONFIGURATION DAEMON

The purpose of the configuration daemon program is to establish and maintain a connection between the IOC multiplexer driver module and the EP driver module. The configuration daemon program is called `epioc`. Its source code, contained in the file `epioc.c`, is shown below.

```
#include <fcntl.h>
#include <stropts.h>

main()
{
  int fd_ep;
  int fd_ioc;

  switch(fork()) {
    case 0: {
      break;
    }
    case -1: {
      perror("fork failed");
      exit(0);
    }
    default: {
      exit(0);
    }
  }
  (void)setpgrp();

  if((fd_ep = open("/dev/ep0", O_RDWR)) < 0) {
    perror("open of /dev/ep0 failed");
    exit(0);
  }

  if((fd_ioc = open("/dev/ioc0", O_RDWR)) < 0) {
    perror("open of /dev/ioc0 failed");
    exit(0);
  }
```

```
    if(ioctl(fd_ioc, I_POP, 0) < 0) {
      perror("I_POP ioctl failed");
      exit(0);
    }

    if(ioctl(fd_ioc, I_LINK, fd_ep) < 0) {
      perror("I_LINK ioctl failed");
      exit(0);
    }

    (void)close(fd_ep);

    (void)pause();
}
```

A daemon program, by definition, is a program that executes in the background to provide a specific service as needed. In the case of `epioc`, the service does not involve actual processing once the link between the IOC multiplexer driver module and the EP driver module has been established. The link is maintained as long as the `epioc` program remains as a non-terminated process in the Unix kernel process table. The `epioc` program uses the `pause` system call to place itself permanently into the sleeping state without terminating.

An obvious problem of using the shell to start a program that lasts forever is regaining control of the shell after starting the program. One popular solution appends an ampersand symbol, `&`, at the end of the shell command line. The first few lines of `epioc.c` show an alternative approach that avoids the need for the `&`, although no damage is done if one is inadvertently appended. The program spawns another execution of itself using the `fork` system call. The parent program, which sees the `fork` system call return a positive non-zero process identifier, immediately exits. This returns control back to the shell. Assuming that the `fork` system call executes successfully, the child program, which sees the `fork` system call return `0`, executes the remainder of the program which executes forever.

The `epioc` program opens both the IOC multiplexer driver module port dedicated to the `epioc` program and the EP driver module port to obtain a file descriptor for each. The file descriptors identify the modules to be linked.

AN/UYK-44(V) IOC channel emulations need an IOA module between the IOC multiplexer driver module port and the stream head. Rather than having the applications explicitly push the IOA module into the stream every time they open an IOC multiplexer driver module port, the IOC multiplexer driver module automatically performs the push when the port is opened. The `epioc` program, on the other hand, neither needs nor desires an IOA module. Unfortunately, there is no convenient way to prevent this since the IOC multiplexer driver module treats all ports, including the port dedicated to the `epioc` program, identically. The `epioc` program, therefore, pops the IOA module from the stream after opening its dedicated IOC multiplexer driver module port.

The epioc program employs the ioctl system call with the I_LINK parameter to link the EP driver module to the IOC multiplexer driver module. Once the linking has been completed, the IOC multiplexer driver module no longer needs the EP driver module file descriptor. The EP driver module port could be left open. Closing it, however, has the advantage of freeing the operating system data structure associated with the file descriptor for other applications. Finally, the epioc program calls the pause system call to enter the sleep state as described previously.

4.2 LOADABLE MODULES

The IOA module, the IOC multiplexer driver module, and the EP driver module are all loadable modules. Loadable modules do not require an operating system reboot when they are installed or replaced.

As described previously, the configuration daemon program, epioc, maintains a connection between the IOC multiplexer driver module and the EP driver module. This program must be killed before replacing a module. The following shell program, contained in the file kill_epioc, kills any epioc program it finds in the Unix kernel process table.

```
KILL=epioc
N='ps aux | grep -w $KILL | cut -c10-14'
for value in $N
do
  if [ $$ -gt $value ]; then
    kill -9 $value
  fi
done
```

The following shell program, located in the file load_ioa, unloads any existing IOA module and loads a new one.

```
if [ 'modstat | grep -w ioa | wc -l' -gt "0" ]; then
  modunload -id 'modstat | grep -w ioa | cut -c1-3'
fi
modload ioa.o -entry _ioainit
chmod 600 ioa
```

A similar shell program, located in the file load_ioc, performs the same function for the IOC multiplexer driver module.

```
if [ 'modstat | grep -w ioc | wc -l' -gt "0" ]; then
  modunload -id 'modstat | grep -w ioc | cut -c1-3'
fi
modload ioc.o -entry _iocinit -exec install_ioc
chmod 600 ioc
```

An important difference between the shell programs in the files `load_ioa` and `load_ioc` is that file `load_ioc` specifies the `-exec` parameter on its `modload` command while file `load_ioa` does not. Unlike the IOA module, the IOC multiplexer driver module is a driver. Unix drivers have a major device number. They also have one or more entries in the `/dev` directory associating file names with minor device numbers for that major device number. The `/dev` directory entries cannot be generated until the major device number is known. The `modload` command selects the major device number. The `-exec` parameter of the `modload` command passes the major device number for character devices as parameter `$4` to the shell command immediately following `-exec` on the command line. The shell command receiving the major device number, in turn, generates the needed `/dev` directory entries. In this case, the shell command receiving the major device number is the shell program contained in the file `install_ioc`.

The shell program in file `install_ioc` contains two nested loops that create `/dev` directory entries for the sixteen channels provided by each of the four input/output controllers. It also creates the `/dev` directory entry for the port dedicated to the `epioc` program.

```
DEV=/dev/ioc
rm -f $DEV\0
/etc/mknod $DEV\0 c $4 0
chmod 666 $DEV\0
minor=1
for ctlr in 0 1 2 3
do
  for chan in 0 1 2 3 4 5 6 7 8 9 a b c d e f
  do
    rm -f $DEV$ctlr.$chan
    /etc/mknod $DEV$ctlr.$chan c $4 $minor
    chmod 666 $DEV$ctlr.$chan
    minor='expr $minor + 1'
  done
done
```

The shell program for the EP driver module, contained in the file `load_ep`, has the same form as that for the IOC multiplexer driver module.

```
if [ 'modstat | grep -w ep | wc -l' -gt "0" ]; then
  modunload -id 'modstat | grep -w ep | cut -c1-3'
fi
modload ep.o -entry _epinit -conf ep.conf -exec install_ep
chmod 600 ep
```

Since the EP driver module is not a multiplexer, the `install_ep` file is much simpler.

```
rm -f /dev/ep0
mknod /dev/ep0 c $4 0
chmod 666 /dev/ep0
```

The `Makefile` to install the first instance, or to replace an existing instance of any of the three modules, is as follows:

```
CFLAGS=-DDEBUG=0 -O -Dsun4 -DKERNEL -DVDDRV -I/sys -target sun4 -c

all: ioa ioc ep epioc

ioa: ioa.o ./load_ioa
        ./kill_epioc
        ./load_ioa
        ./epioc

ioc: ioc.o ./load_ioc ./install_ioc
        ./kill_epioc
        ./load_ioc
        ./epioc

ep: ep.o ./load_ep ./install_ep ./ep.conf
        ./kill_epioc
        ./load_ep
        ./epioc

ioa.o: ioa.c epmsg.h epcntl.h
        cc $(CFLAGS) ioa.c

ioc.o: ioc.c epmsg.h
        cc $(CFLAGS) ioc.c

ep.o: ep.c epmsg.h epreg.h
        cc $(CFLAGS) ep.c

epioc: epioc.c
        cc -o epioc epioc.c
        ./kill_epioc
        ./epioc
```

4.3 HARDWARE PARAMETERS

The EP driver module interfaces with the VME RAM board. Unix needs to know the VMEbus starting address and the VMEbus access mode of the VME RAM board so that it can map it into the virtual address space seen by the EP driver module. The EP driver module receives interrupts from the VME AN/UYK-44(V) EP board. Unix needs to know the VMEbus interrupt priority and the VMEbus interrupt acknowledge vector of the interrupt generated by

the VME AN/UYK−44(V) EP board so it can direct its interrupts to the interrupt handler within the EP driver module.

All this information is provided by the file `ep.conf` that Unix reads when it loads the EP driver module.

```
device ep0 at vme32d32 ? csr 0x10000000 priority 2 vector epintr 0xe4
```

The file contents are interpreted in the following manner. The EP driver module interfaces to a real hardware `device`. There is only one such device, and it is identified as `ep0`. The VME RAM board supports VMEbus access mode A32 D32. The VME RAM board has a VMEbus starting address of `0x10000000`. The interrupt generated by the VME AN/UYK−44(V) EP board has VMEbus interrupt priority 2 and interrupt acknowledge vector `0xe4`. Unix is to direct these interrupts to the interrupt handler located at address `epintr` within the EP driver module.

Since Unix does not need to know the number of memory bytes in the VME RAM board, that information does not appear in the `ep.conf` file. The VME RAM board size is important to the EP driver module. The size is specified in terms of 16−bit AN/UYK−44(V) words within the EP driver module file `epreg.h`.

4.4 SYSTEM BOOT

To automatically start the AN/UYK−44(V) IOC emulation when the system boots, add the following line to the `/etc/rc.local` file if it is does not already exist.

```
/usr/local/ep/rc.ep
```

The AN/UYK−44(V) EP initialization shell file `/usr/local/ep/rc.ep` contains the following commands.

```
echo "UYK-44EP Initialization"
DIR=/usr/local/ep

$DIR/kill_epioc

$DIR/load_ep
$DIR/load_ioc
$DIR/load_ioa

$DIR/epioc
```

The call to the shell file `kill_epioc` is not really necessary during reboot, but is included anyway for safety reasons.

5.0 EP DRIVER MODULE SOURCE LISTINGS

5.1 AN/UYK-44(V) MEMORY BOARD STRUCTURE - epreg.h

```
#define EP_MEM_SIZE (1024*1024)
#define RUPT44_PARAM_BLK_PTR (EP_MEM_SIZE-256-10)

struct ep_mem {
  unsigned short ep_word[EP_MEM_SIZE-256-12];
  unsigned long ep_mailbox_ptr;
  unsigned short ep_i_lock;
  unsigned short ep_class;
  unsigned short ep_code;
  unsigned short ep_sr2;
  unsigned short ep_c_lock;
  unsigned short ep_limit;
  unsigned short ep_iocr;
  unsigned short ep_rega;
  unsigned short ep_cell[2];
  unsigned short ep_page_reg[256];
};
```

5.2 STREAM MESSAGE FORMATS - epmsg.h

```
struct msg_cntl_part {
  unsigned char  function;
  unsigned char  ack;
  unsigned char  device;
  unsigned char  chain;
  unsigned short address;
  unsigned short data;
  unsigned short control;
  unsigned short status;
  unsigned short limit;
};

/* ack field definitions */

#define DISCARD          0      /* discard message after execution */
#define REPLY            1      /* return message after execution */

/* chain field definitions */

#define OUTPUT           0      /* output chain */
#define INPUT            1      /* input chain */
```

```
/* function, address, data, and control field definitions */

#define NO_FUNCTION      0

#define IO_CELL          1      /* IO command cell                    */
                                /*  address: second command cell word */
                                /*  data:    not used                 */
                                /*  control: first command cell word  */

#define READ_INST        2      /* read next instruction              */
                                /*  address: instruction address      */
                                /*  data:    instruction contents     */
                                /*  control: not used                 */

#define READ_INST_Y      3      /* read instruction y field           */
                                /*  address: y field address          */
                                /*  data:    y field contents         */
                                /*  control: instruction contents     */

#define JUMP_INST        4      /* jump and read next instruction     */
                                /*  address: instruction address      */
                                /*  data:    instruction contents     */
                                /*  control: not used                 */

#define JUMP_STAT        5      /* jump if status bit set             */
                                /*    and read next instruction       */
                                /*  address: instruction address      */
                                /*  data:    instruction contents     */
                                /*  control: status register mask     */

#define READ_REG         6      /* read control register              */
                                /*  address: not used                 */
                                /*  data:    register contents        */
                                /*  control: register number          */

#define WRITE_REG        7      /* write control register             */
                                /*  address: not used                 */
                                /*  data:    write data               */
                                /*  control: register number          */

#define REG_TO_MEM       8      /* copy control register to memory    */
                                /*  address: memory address           */
                                /*  data:    register contents        */
                                /*  control: register number          */

#define MEM_TO_REG       9      /* copy memory to control register    */
                                /*  address: memory address           */
                                /*  data:    memory contents          */
                                /*  control: register number          */
```

```
#define READ_START      10      /* read BCW and BAP                    */
                                /*  address: buffer address pointer    */
                                /*  data:    address of BCW-BAP        */
                                /*  control: buffer control word       */

#define READ_BUF        11      /* read memory buffer                  */
                                /*  address: buffer address pointer    */
                                /*  data:    not used                  */
                                /*  control: buffer control word       */

#define WRITE_START     12      /* write BCW and BAP                   */
                                /*  address: buffer address pointer    */
                                /*  data:    address of BCW-BAP        */
                                /*  control: buffer control word       */

#define WRITE_BUF       13      /* write memory buffer                 */
                                /*  address: buffer address pointer    */
                                /*  code:    not used                  */
                                /*  control: buffer control word       */

#define EF_START        14      /* read external function BCW and BAP  */
                                /*  address: buffer address pointer    */
                                /*  data:    address of BCW-BAP        */
                                /*  control: buffer control word       */

#define EF_BUF          15      /* read external function memory buffer */
                                /*  address: buffer address pointer    */
                                /*  data:    not used                  */
                                /*  control: buffer control word       */

#define WRITE_WORD      16      /* write memory word thru mask         */
                                /*  address: memory address            */
                                /*  data:    memory data               */
                                /*  control: write mask                */

#define INTERRUPT       17      /* generate UYK-44 interrupt           */
                                /*  address: interrupt code            */
                                /*  data:    interrupt class           */
                                /*  control: CP status register 2      */

#define STAT_TO_MEM     18      /* copy status register to memory      */
                                /*  address: memory address            */
                                /*  data:    status register contents  */
                                /*  control: status register number    */

#define WRITE_STAT      19      /* write status register               */
                                /*  address: status regieter number    */
                                /*  data:    write data                */
                                /*  control: write mask                */
```

```
#define TEST_WORD       20      /* test memory word thru mask          */
                                /*  address: memory address            */
                                /*  data:    memory read data          */
                                /*  control: memory data mask          */

#define TEST_AND_SET    21      /* test and set memory word thru mask  */
                                /*  address: memory address            */
                                /*  data:    memory read/write data    */
                                /*  control: memory data mask          */

#define CHAN_CONTROL    22      /* channel control                     */
                                /*  address: channel select bits       */
                                /*  data:    control bits set          */
                                /*  control: control bit clear mask    */

#define WRITE_EI_WORD   23      /* write external interrupt word       */
                                /*  address: 32-bit EI word extension  */
                                /*  data:    EI word data              */
                                /*  control: size of EI word in bytes  */

/* status field definitions */

#define EP_NO_FAULTS    0       /* EP memory access ok */
#define EP_READ_FAULT   1       /* EP memory read protect fault        */
#define EP_WRITE_FAULT  2       /* EP memory write protect fault       */
#define EP_EXEC_FAULT   3       /* EP memory execute protect fault     */
#define EP_RANGE_FAULT  4       /* EP memory address range fault       */
```

5.3 IOC REGISTER DEFINITIONS – epreg.h

```
/* control register definitions */

#define IN_BCW          0x0     /* input buffer control word */
#define IN_BAP          0x1     /* input buffer address pointer */
#define IN_CAP          0x2     /* input chain address pointer */
#define OUT_BCW         0x4     /* output buffer control word */
#define OUT_BAP         0x5     /* output buffer address pointer */
#define OUT_CAP         0x6     /* output chain address pointer */

/* IOC status register 0 bit definitions */

#define IN_RUPT_PEND    0x0100  /* input chain interrupt pending */
#define OUT_RUPT_PEND   0x0200  /* output chain interrupt pending */
#define EI_RUPT_PEND    0x0400  /* external interrupt pending */
#define ICTO_RUPT_PEND  0x0800  /* intercomputer time-out interrupt pending */
#define IN_BUF_ACTIVE   0x1000  /* input chain active */
#define OUT_BUF_ACTIVE  0x2000  /* output chain active */
#define EI_WORD_PEND    0x4000  /* external interrupt enable */
#define COND_TEST       0x8000  /* conditional jump test bit */
```

```
/* IOC status register 3 bit definitions */

#define IN_CH_ACTIVE    0x0001  /* active input chain flag */
#define IN_CH_ABORT     0x0002  /* abort input chain flag */
#define OUT_CH_ACTIVE   0x0004  /* active output chain flag */
#define OUT_CH_ABORT    0x0008  /* abort output chain flag */
#define EIE_LINE        0x0010  /* external interrupt data enable */
#define RUPT_GEN        0x0020  /* interrupt generation enable */

/* flag definitions */

#define BF_FLAG         0xC000  /* biased fetch flag mask */
#define ALL_ONES        0xFFFF  /* all flag bits set to one */
#define ALL_ZEROS       0x0000  /* all flag bits set to zero */
```

5.4 AN/UYK-44(V) INTERRUPT DEFINITIONS – epreg.h

```
#define CLASS_1         1       /* class 1 interrupt */

#define RESUME_FAULT    0x2     /* memory resume fault */

#define CLASS_2         2       /* class 2 interrupt */

#define COMMAND_FAULT   0x02    /* IOC command instruction fault */

#define PROTECT_FAULT   0x18    /* memory protection fault */

#define CLASS_3         3       /* class 3 interrupt */
```

5.5 GLOBAL DECLARATIONS

```
#ifndef DEBUG
# define DEBUG 0
#endif

#include "ep.h"
#include "epreg.h"
#include "epmsg.h"#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/dir.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/errno.h>

#include <sys/buf.h>
#include <sys/conf.h>
#include <sun/vddrv.h>
```

```
/* driver related declarations */

#include <sundev/mbvar.h>

int epprobe();                  /* VME probe routine */

struct mb_device *epdinfo[NEP];

struct mb_driver epdriver = {
    epprobe,                    /* probe */
    0,                          /* slave */
    0,                          /* attach */
    0,                          /* go */
    0,                          /* done */
    0,                          /* intr */
    sizeof(struct ep_mem),      /* size */
    "ep",                       /* device name */
    epdinfo,                    /* device state struct */
    0,                          /* controller name */
    0,                          /* controller state struct */
    0,                          /* flags */
    0                           /* interrupt linked list */
};

/* stream related declarations */

static int epopen();    /* open routine, called on each open */
static int epwput();    /* write queue put procedure */
static int epwsvr();    /* write queue service routine */
static int epclose();   /* close routine, called on last close */

static struct module_info minfo = {
    0,                  /* ID number */
    "ep",               /* name */
    0,                  /* queue minimum packet size accepted */
    INFPSZ,             /* queue maximum packet size accepted */
    150,                /* queue flow control high water mark */
    50                  /* queue flow control low water mark */
};

static struct qinit rinit = {
    NULL,               /* read queue put routine */
    NULL,               /* read queue service routine */
    epopen,             /* open routine, called on each open */
    epclose,            /* close routine, called on last close */
    NULL,               /* reserved */
    &minfo,             /* pointer to read queue information structure */
    NULL                /* pointer to read queue statistics structure */
};
```

```
static struct qinit winit = {
    epwput,             /* write queue put routine */
    epwsvr,             /* write queue service routine */
    NULL,               /* ignored */
    NULL,               /* ignored */
    NULL,               /* reserved */
    &minfo,             /* pointer to write queue information structure */
    NULL                /* pointer to write queue statistics structure */
};

struct streamtab epinfo = {
    &rinit,             /* pointer to read queue initialization structure */
    &winit,             /* pointer to write queue initialization structure */
    NULL,               /* multiplexer drivers only - not used */
    NULL,               /* multiplexer drivers only - not used */
    NULL                /* pointer to module push list on first open */
};

struct ep {
    queue_t *rq;        /* pointer to read queue */
    queue_t *wq;        /* pointer to write queue */
    struct ep_mem *mem; /* pointer to UYK-44 memory */
    char openflg;       /* device open flag */
    char errflg;        /* memory access error flag */
};
struct ep ep_ep[NEP];   /* minor device private data structures */
int ep_cnt = NEP;       /* number of valid minor devices */
```

5.6 LOADABLE MODULE STRUCTURES

```
extern int nulldev();
extern int nodev();

struct cdevsw epcdev = {
    nodev, nodev, nodev, nodev, nodev, nodev,
    nodev, 0, &epinfo, 0,
};

struct vdldrv epvd = {
    VDMAGIC_DRV,        /* type of module is device */
    "ep",               /* name of device */
    NULL,               /* mb_ctlr structure address */
    &epdriver,          /* mb_driver structure address */
    NULL,               /* mb_device array structure address, use conf file */
    0,                  /* number of controllers */
    NEP,                /* number of devices */
    NULL,               /* bdevsw entry address */
```

```
    &epcdev,            /* cdevsw entry address */
    0,                  /* block device number chosen by system */
    0                   /* character device number chosen by system */
};
```

5.7 LOADABLE MODULE INITIALIZATION

```
epinit(cmd_code, vdp, vdi, vds)
unsigned int cmd_code;
struct vddrv *vdp;
caddr_t vdi;
struct vdstat vds;
{

  switch(cmd_code) {
    case VDLOAD: {
      vdp->vdd_vdtab = (struct vdlinkage *)&epvd;
      return(0);
    }
    case VDUNLOAD: {
      return(0);
    }
    case VDSTAT: {
      return(0);
    }
    default: {
      return(EIO);
    }
  }
}
```

5.8 PROBE ROUTINE

```
epprobe(addr, unit)
caddr_t addr;                   /* hardware kernel virtual address */
int unit;                       /* minor device number */
{
  register struct ep_mem *ep_mem;
  register int i;

  ep_mem = (struct ep_mem *)addr;

  if(peekc((char *)&ep_mem->ep_word[0]) == -1) {
    return(0);
  }
  if(peekc((char *)&ep_mem->ep_page_reg[255]) == -1) {
```

```
    printf("ep: bad memory board address range\n");
    return(0);
  }

  for(i = 0; i < 256; i++) {
    if(poke((short *)&ep_mem->ep_page_reg[i], i)) {
      printf("ep: IOC page register initialization failure\n");
      return(0);
    }
  }

  return(sizeof(struct ep_mem));
}
```

5.9 INTERRUPTER ROUTINE

```
static int
rupt44(ep, class, code, sr2)
struct ep *ep;
unsigned short class;
unsigned short code;
unsigned short sr2;
{
  struct ep_mem *ep_mem;
  unsigned long absaddr;
  unsigned long *p;

#if DEBUG > 0
  printf("ep: UYK-44 Interrupt Class %x Code %x SR2 %x\n", class, code, sr2);
#endif

  ep_mem = ep->mem;
  if(ep_mem->ep_i_lock != 0) {
    printf("ep: UYK-44 interrupt parameter block locked\n");
    return(0);
  }
  ep_mem->ep_i_lock = 0xFFFF;

  ep_mem->ep_class = class;
  ep_mem->ep_code = code;
  ep_mem->ep_sr2 = sr2;

  absaddr = ep_mem->ep_mailbox_ptr;
  if(absaddr >= (sizeof(ep_mem->ep_word) >> 1)) {
    printf("ep: bad ep_mailbox_ptr\n");
    return(0);
  }
  p = (unsigned long *)(&ep_mem->ep_word[absaddr]);
```

```
  *p = RUPT44_PARAM_BLK_PTR;
  return(1);
}
```

5.10 READ AN/UYK-44(V) INSTRUCTION WORD ROUTINE

```
static unsigned short
ri44w(ep, pageset, reladdr)
struct ep *ep;
int pageset;
unsigned short reladdr;          /* UYK-44 relative address */
{
  struct ep_mem *ep_mem;
  int pagenum;
  unsigned long absaddr;

  ep_mem = ep->mem;
  pagenum = ((pageset & 0x3) << 6) + ((reladdr >> 10) & 0x3F);
  if(ep_mem->ep_page_reg[pagenum] & 0x4000) {
    ep->errflg = EP_EXEC_FAULT;
    rupt44(ep, CLASS_2, PROTECT_FAULT, 0);
    return(0);
  }
  absaddr = (reladdr & 0x3FF) + ((ep_mem->ep_page_reg[pagenum] & 0xFFF) <<
10);
  if(absaddr >= (sizeof(ep_mem->ep_word) >> 1)) {
    ep->errflg = EP_RANGE_FAULT;
    rupt44(ep, CLASS_1, RESUME_FAULT, 0);
    return(0);
  }
  return(ep_mem->ep_word[absaddr]);
}
```

5.11 READ AN/UYK-44(V) DATA WORD ROUTINE

```
static unsigned short
rd44w(ep, pageset, reladdr)
struct ep *ep;
int pageset;
unsigned short reladdr;          /* UYK-44 relative address */
{
  struct ep_mem *ep_mem;
  int pagenum;
  unsigned long absaddr;

  ep_mem = ep->mem;
  pagenum = ((pageset & 0x3) << 6) + ((reladdr >> 10) & 0x3F);
  if(ep_mem->ep_page_reg[pagenum] & 0x1000) {
    ep->errflg = EP_READ_FAULT;
    rupt44(ep, CLASS_2, PROTECT_FAULT, 0);
    return(0);
```

```
  }
  absaddr = (reladdr & 0x3FF) + ((ep_mem->ep_page_reg[pagenum] & 0xFFF) <<
10);
  if(absaddr >= (sizeof(ep_mem->ep_word) >> 1)) {
    ep->errflg = EP_RANGE_FAULT;
    rupt44(ep, CLASS_1, RESUME_FAULT, 0);
    return(0);
  }
  return(ep_mem->ep_word[absaddr]);
}
```

5.12 WRITE AN/UYK-44(V) DATA WORD ROUTINE

```
static void
wd44w(ep, pageset, reladdr, value)
struct ep *ep;
int pageset;
unsigned short reladdr;          /* UYK-44 relative address */
unsigned short value;            /* write data */
{
  struct ep_mem *ep_mem;
  int pagenum;
  unsigned long absaddr;

  ep_mem = ep->mem;
  pagenum = ((pageset & 0x3) << 6) + ((reladdr >> 10) & 0x3F);
  if(ep_mem->ep_page_reg[pagenum] & 0x2000) {
    ep->errflg = EP_WRITE_FAULT;
    rupt44(ep, CLASS_2, PROTECT_FAULT, 0);
    return;
  }
  absaddr = (reladdr & 0x3FF) + ((ep_mem->ep_page_reg[pagenum] & 0xFFF) <<
10);
  if(absaddr >= (sizeof(ep_mem->ep_word) >> 1)) {
    ep->errflg = EP_RANGE_FAULT;
    rupt44(ep, CLASS_1, RESUME_FAULT, 0);
    return;
  }
  ep_mem->ep_word[absaddr] = value;
  ep_mem->ep_page_reg[pagenum] |= 0x8000;
}
```

5.13 WRITE AN/UYK-44(V) DATA BYTE ROUTINE

```
static void
wd44b(ep, pageset, reladdr, bflag, value)
```

```
struct ep *ep;
int pageset:
unsigned short reladdr;          /* UYK-44 relative address */
char bflag;                      /* 0 for upper byte, 1 for lower byte */
unsigned short value;            /* write data */
{
  struct ep_mem *ep_mem;
  int pagenum;
  unsigned long absaddr;
  unsigned char *p;

  ep_mem = ep->mem;
  pagenum = ((pageset & 0x3) << 6) + ((reladdr >> 10) & 0x3F);
  if(ep_mem->ep_page_reg[pagenum] & 0x2000) {
    ep->errflg = EP_WRITE_FAULT;
    rupt44(ep, CLASS_2, PROTECT_FAULT, 0);
    return;
  }
  absaddr = (reladdr & 0x3FF) + ((ep_mem->ep_page_reg[pagenum] & 0xFFF) <<
10);
  if(absaddr >= (sizeof(ep_mem->ep_word) >> 1)) {
    ep->errflg = EP_RANGE_FAULT;
    rupt44(ep, CLASS_1, RESUME_FAULT, 0);
    return;
  }
  p = (unsigned char *)&ep_mem->ep_word[absaddr];
  if(bflag) {
    p++;
  }
  *p = value;
  ep_mem->ep_page_reg[pagenum] |= 0x8000;
}
```

5.14 READ–MODIFY–WRITE AN/UYK–44(V) DATA WORD ROUTINE

```
#ifdef EPRMW
extern unsigned short rmw();
#else
unsigned short
rmw(addr, wrdata, mask)
unsigned short *addr;
unsigned short wrdata;
unsigned short mask;
{
  register unsigned short rddata;

  rddata = *addr;
  if(mask) {
```

```
    *addr = (wrdata & mask) + (rddata & ~mask);
  }
  return(rddata);
}
#endif

static unsigned short
rmw44w(ep, pageset, reladdr, value, mask)
struct ep *ep;
int pageset;
unsigned short reladdr;          /* UYK-44 relative address */
unsigned short value;            /* write data */
unsigned short mask;             /* write data mask */
{
  struct ep_mem *ep_mem;
  int pagenum;
  unsigned long absaddr;
  unsigned short readval;

  ep_mem = ep->mem;
  pagenum = ((pageset & 0x3) << 6) + ((reladdr >> 10) & 0x3F);
  if(ep_mem->ep_page_reg[pagenum] & 0x1000) {
    ep->errflg = EP_READ_FAULT;
    goto dorupt;
  }
  if(ep_mem->ep_page_reg[pagenum] & 0x2000) {
    ep->errflg = EP_WRITE_FAULT;
  dorupt:
    rupt44(ep, CLASS_2, PROTECT_FAULT, 0);
    return(0);
  }
  absaddr = (reladdr & 0x3FF) + ((ep_mem->ep_page_reg[pagenum] & 0xFFF) <<
10);
  if(absaddr >= (sizeof(ep_mem->ep_word) >> 1)) {
    ep->errflg = EP_RANGE_FAULT;
    rupt44(ep, CLASS_1, RESUME_FAULT, 0);
    return(0);
  }
  readval = rmw(&ep_mem->ep_word[absaddr], value, mask);
  ep_mem->ep_page_reg[pagenum] |= 0x8000;
  return(readval);
}
```

5.15 GET BUFFER LENGTH ROUTINE

```
getbcnt(bcw)
unsigned short bcw;
```

```
{
  int count;

  count = bcw & 0x0FFF;
  if(count == 0) {
    count = 0x1000;
  }
  if((bcw & 0xC000) == 0xC000) {
    return(count << 2);
  }
  if((bcw & 0xC000) == 0x8000) {
    return(count << 1);
  }
  return(count);
}
```

5.16 ADJUST BUFFER LENGTH ROUTINE

```
void
adjbcnt(pbcw, count)
unsigned short *pbcw;
int count;
{
  if(count & 1) {
    *pbcw ^= 0x1000;
  }
  if((*pbcw & 0xC000) == 0xC000) {
    count >>= 2;
  }
  else if((*pbcw & 0xC000) == 0x8000) {
    count >>= 1;
  }
  *pbcw = (((*pbcw | 0x1000) - count) & 0x0FFF) + (*pbcw & 0xF000);
}
```

5.17 SET BUFFER PAGE SET ROUTINE

```
bufpset(bcw, chan)
unsigned short bcw;
unsigned char chan;
{
  if(bcw & 0x2000) {
    if(chan & 0x08) {
      return(3);
    }
    return(2);
```

```
  }
  return(0);
}
```

5.18 OPEN ROUTINE

```
static int
epopen(q, dev, flag, sflag)
queue_t *q;                     /* pointer to read queue */
dev_t dev;                      /* major and minor device numbers */
int flag;                       /* file open flags */
int sflag;                      /* stream open flags */
{
  struct ep *ep;

  if(sflag) {                   /* check for normal driver open */
    return(OPENFAIL);
  }

  dev = minor(dev);             /* extract minor device number */

  if(dev >= ep_cnt) {           /* check minor device number range */
    return(OPENFAIL);
  }

  if(q->q_ptr) {                /* check to see if already open */
    u.u_error = EBUSY;
    return(OPENFAIL);
  }

  ep = &ep_ep[dev];             /* pointer to minor device private structure
*/

  ep->rq = q;                   /* pointer to read queue */
  ep->wq = WR(q);               /* pointer to write queue */

  q->q_ptr = (char *)ep;        /* read queue link */
  WR(q)->q_ptr = (char *)ep;    /* write queue link */

  ep->mem = (struct ep_mem *)epdinfo[dev]->md_addr;
                                /* pointer to UYK-44 memory */

  ep->openflg = 1;              /* mark device opened */
  return(dev);
}
```

5.19 WRITE PUT ROUTINE

```
static int
epwput(q, mp)
queue_t *q;                     /* pointer to write queue */
mblk_t *mp;                     /* pointer to message */
{

  switch(mp->b_datap->db_type) {

    default: {
      freemsg(mp);                          /* discard message */
      break;
    }

    case M_FLUSH: {
      if(*mp->b_rptr & FLUSHW) {            /* if flush write flag set */
        flushq(q, FLUSHDATA);               /* flush write queue */
      }
      if(*mp->b_rptr & FLUSHR) {            /* if flush read flag set */
        *mp->b_rptr &= ~FLUSHW;             /* clear flush write flag */
        qreply(q, mp);                      /* send message back upstream */
      }
      else {                                /* if flush read flag not set */
        freemsg(mp);                        /* discard message */
      }
      break;
    }

    case M_PROTO:
    case M_PCPROTO:
    case M_IOCTL: {
      putq(q, mp);                          /* put message on driver write queue
*/
      break;
    }
  }
  return(0);
}
```

5.20 WRITE QUEUE SERVICE ROUTINE

```
static int
epwsvr(q)
queue_t *q;
{
  mblk_t *mp;
```

```
while((mp = getq(q)) != NULL) {
  switch(mp->b_datap->db_type) {

    case M_PROTO:
    case M_PCPROTO: {
      struct ep *ep;
      struct msg_cntl_part *p;
      unsigned char reply_flag;

      ep = (struct ep *)q->q_ptr;      /* minor device private structure */
      p = (struct msg_cntl_part *)(mp->b_datap->db_base);

      ep->errflg = EP_NO_FAULTS;
      reply_flag = p->ack;

      switch(p->function) {

        case READ_INST:
        case READ_INST_Y:
        case JUMP_INST:
        case JUMP_STAT: {
          reply_flag = REPLY;
          p->data = ri44w(ep, 0, p->address);
          break;
        }

        case MEM_TO_REG:
        case TEST_WORD: {
          reply_flag = REPLY;
          p->data = rd44w(ep, 0, p->address);
          break;
        }

        case REG_TO_MEM:
        case STAT_TO_MEM: {
          wd44w(ep, 0, p->address, p->data);
          if(ep->errflg != EP_NO_FAULTS) {
            reply_flag = REPLY;
          }
          break;
        }

        case WRITE_WORD:
        case TEST_AND_SET: {
          reply_flag = REPLY;
          p->data = rmw44w(ep, 0, p->address, p->data, p->control);
          break;
        }

        case READ_START:
        case WRITE_START:
```

```
case EF_START: {
  reply_flag = REPLY;
  p->control = rd44w(ep, 0, p->data);          /* get BCW */
  p->address = rd44w(ep, 0, p->data + 1);      /* get BAP */
  break;
}

case READ_BUF:
case EF_BUF: {
  mblk_t *bp;                  /* pointer to data block */
  int pageset;                 /* buffer page set */
  int bcount;                  /* number of buffer bytes */
  int count;                   /* number of buffer bytes left */
  unsigned short addr;         /* 16-bit word address */
  unsigned short rddata;       /* 16-bit word data */

  reply_flag = REPLY;
  pageset = bufpset(p->control, p->device);
  bcount = count = getbcnt(p->control);
  if((bp = allocb(count, BPRI_MED)) == NULL) {
    printf("ep: epwsvr: allocb failed\n");
    break;
  }
  addr = p->address;
  if(count > 0 && (p->control & 0xD000) == 0x5000) {
    rddata = rd44w(ep, pageset, addr);
    if(ep->errflg != EP_NO_FAULTS) {
      goto rbabort;
    }
    *bp->b_wptr++ = (unsigned char)rddata;
    count--;
    addr++;
  }
  while(count >= 2) {
    rddata = rd44w(ep, pageset, addr);
    if(ep->errflg != EP_NO_FAULTS) {
      goto rbabort;
    }
    *bp->b_wptr++ = (unsigned char)(rddata >> 8);
    *bp->b_wptr++ = (unsigned char)rddata;
    count -= 2;
    addr++;
  }
  if(count == 1) {
    rddata = rd44w(ep, pageset, addr);
    if(ep->errflg != EP_NO_FAULTS) {
      goto rbabort;
    }
```

```
      *bp->b_wptr++ = (unsigned char)(rddata >> 8);
      count = 0;
    }
rbabort:
    p->address = addr;
    adjbcnt(&(p->control), bcount - count);
    linkb(mp, bp);
    break;
}

case WRITE_BUF: {
    mblk_t *bp;                 /* pointer to data block */
    mblk_t *bp_next;            /* pointer to next data block */
    int pageset;                /* buffer page set */
    int bcount;                 /* number of buffer bytes */
    int count;                  /* number of buffer bytes left */
    unsigned short addr;        /* 16-bit word address */
    unsigned short wrdata;      /* 16-bit word data */
    int size;                   /* message buffer size in bytes */

    reply_flag = REPLY;
    for(bp = mp->b_cont; bp != NULL; bp = bp_next) {
      if(!ep->errflg) {
        bp_next = bp->b_cont;
        pageset = bufpset(p->control, p->device);
        bcount = count = getbcnt(p->control);
        size = bp->b_wptr - bp->b_rptr;
        if(size < count) {
          count = size;
        }
        addr = p->address;
        if(count > 0 && (p->control & 0xD000) == 0x5000) {
          wrdata = (unsigned short)(*bp->b_rptr++);
          wd44b(ep, pageset, addr, 1, wrdata);
          if(ep->errflg != EP_NO_FAULTS) {
            goto wbabort;
          }
          count--;
          addr++;
        }
        while(count >= 2) {
          wrdata = (unsigned short)(*bp->b_rptr++) << 8;
          wrdata += (unsigned short)(*bp->b_rptr++);
          wd44w(ep, pageset, addr, wrdata);
          if(ep->errflg != EP_NO_FAULTS) {
            goto wbabort;
          }
          count -= 2;
```

```
            addr++;
          }
          if(count == 1) {
            wrdata = (unsigned short)(*bp->b_rptr++);
            wd44b(ep, pageset, addr, 0, wrdata);
            if(ep->errflg != EP_NO_FAULTS) {
              goto wbabort;
            }
            count = 0;
          }
        wbabort:
          adjbcnt(&(p->control), bcount - count);
          p->address = addr;
        }
        freeb(bp);
      }
      mp->b_cont = NULL;
      break;
    }

    case WRITE_EI_WORD:
    case INTERRUPT: {
      mblk_t *bp;

      bp = mp;
      for(;;) {
        if(p->function == WRITE_EI_WORD) {
          unsigned short addr;
          addr = 0x80 + (p->device - 1);
          switch(p->control) {
            case 1: {
              wd44w(ep, 0, addr, p->data << 8);    /* upper byte only */
              break;
            }
            case 2: {
              wd44w(ep, 0, addr, p->data);
              break;
            }
            case 4: {
              wd44w(ep, 0, addr | 1, p->address); /* upper 16 bits */
              wd44w(ep, 0, addr & ~1, p->data);   /* lower 16 bits */
              break;
            }
          }
        }
        else if(p->function == INTERRUPT) {
          rupt44(ep, p->data, p->address, p->control);
        }
```

```
            loop:
              bp = bp->b_cont;
              if(bp == NULL) {
                break;
              }
              if(bp->b_datap->db_type != M_PROTO) {
                goto loop;
              }
              p = (struct msg_cntl_part *)(bp->b_datap->db_base);
            }
            break;
          }

          default: {
            break;
          }
        }
        if(reply_flag == DISCARD) {
          freemsg(mp);
        }
        else {
          p->status = ep->errflg;
          qreply(q, mp);
        }
        break;
      }

      case M_IOCTL: {
        mp->b_datap->db_type = M_IOCNAK;
        qreply(q, mp);
        break;
      }
    }
  }
}
```

5.21 CLOSE ROUTINE

```
static int
epclose(q, flag)
queue_t *q;                     /* pointer to read queue */
int flag;                       /* file open flags */
{
  struct ep *ep;

  ep = (struct ep *)q->q_ptr;   /* get minor device private structure */
  ep->openflg = 0;              /* mark device closed */
  return(0);
}
```

5.22 INTERRUPT HANDLER ROUTINE

```
epintr(dev)
int dev;                        /* minor device number */
{
  struct ep *ep;                /* minor device private structure */
  struct ep_mem *ep_mem;
  mblk_t *mp;                   /* pointer to interrupt message */
  register unsigned short t;

  dev = 0;                      /* forced minor device = 0 because bug */
                                /* in modload in SunOS4.1.1. It does */
                                /* not pass dev param thru interrupt */
  ep = &ep_ep[dev];
  ep_mem = ep->mem;

  if(!ep->openflg) {                    /* make sure queue exists */
    printf("ep: unexpected interrupt from UYK-44\n");
    return;
  }

  if((mp = allocb(sizeof(struct msg_cntl_part), BPRI_MED)) == NULL) {
    printf("ep: epintr: allocb failed\n");
    return;
  }

  mp->b_datap->db_type = M_PROTO;

  *mp->b_wptr++ = IO_CELL;              /* load function field */

  *mp->b_wptr++ = REPLY;                /* load ack field */

  t = (ep_mem->ep_cell[0] & 0x00F0) >> 4;
  if(ep_mem->ep_iocr & 0x00F0) {        /* IOC instruction */
    t += ((ep_mem->ep_rega & 0x0003) << 4);
  }
                                        /* load device field */
  *mp->b_wptr++ = (unsigned char)t + 1;

  *mp->b_wptr++ = 0;                    /* load chain field */

  t = ep_mem->ep_cell[1];               /* load address field */
  *mp->b_wptr++ = t >> 8;
  *mp->b_wptr++ = t;

  *mp->b_wptr++ = 0;                    /* load data field */
  *mp->b_wptr++ = 0;

  t = ep_mem->ep_cell[0];               /* load command field */
  *mp->b_wptr++ = t >> 8;
  *mp->b_wptr++ = t;
```

```
    *mp->b_wptr++ = 0;                  /* load status field */
    *mp->b_wptr++ = 0;

    t = ep_mem->ep_limit;               /* load limit field */
    *mp->b_wptr++ = t >> 8;
    *mp->b_wptr++ = t;

    ep_mem->ep_c_lock = 0;              /* clear memory lock */

    putnext(ep->rq, mp);                /* send new message upstream */
}
```

5.23 PRINT MESSAGE DEBUG ROUTINE

```
#if DEBUG > 0
static char *function_str[] = {
    "NO_FUNCTION",      /*  0   no function specified          */
    "IO_CELL",          /*  1   IO command cell                */
    "READ_INST",        /*  2   read instruction               */
    "READ_INST_Y",      /*  3   read instruction y field       */
    "JUMP_INST",        /*  4   jump to new instruction        */
    "JUMP_STAT",        /*  5   jump if status bit set         */
    "READ_REG",         /*  6   read control register          */
    "WRITE_REG",        /*  7   write control register         */
    "REG_TO_MEM",       /*  8   copy control register to memory */
    "MEM_TO_REG",       /*  9   copy memory to control register */
    "READ_START",       /* 10   read BCW and BAP               */
    "READ_BUF",         /* 11   read memory user data buffer   */
    "WRITE_START",      /* 12   write BCW and BAP              */
    "WRITE_BUF",        /* 13   write memory user data buffer  */
    "EF_START",         /* 14   read BCW and BAP               */
    "EF_BUF",           /* 15   read memory user data buffer   */
    "WRITE_WORD",       /* 16   atomic read-modify-write       */
    "INTERRUPT",        /* 17   generate UYK-44 interrupt      */
    "STAT_TO_MEM",      /* 18   copy status register to memory */
    "WRITE_STAT",       /* 19   write status register          */
    "TEST_WORD",        /* 20   test memory word               */
    "TEST_AND_SET",     /* 21   test and set memory word       */
    "CHAN_CONTROL",     /* 22   channel control                */
    "WRITE_EI_WORD"     /* 23   write external interrupt word  */
};

static int
prmsg(mp, s)
mblk_t *mp;
char *s;
{
  struct msg_cntl_part *p;
  mblk_t *bp;
  unsigned short t;

  if(s) {
    printf("%s:\n", s);
  }

  for(bp = mp; bp != NULL; bp = bp->b_cont) {
    switch(bp->b_datap->db_type) {
      case M_PROTO:
      case M_PCPROTO: {
        p = (struct msg_cntl_part *)(bp->b_datap->db_base);
```

```
        t = p->function;
        if(t > 0 && t < (sizeof(function_str) / sizeof(*function_str))) {
          printf("  %s", function_str[t]);
        }
        else {
          printf("  %x", t);
        }
        printf(", ack %s", (p->ack == DISCARD) ? "DISCARD" : "REPLY");
        printf(", device %x", p->device);
        printf(", chain %s\n", (p->chain == OUTPUT) ? "OUTPUT" : "INPUT");
        printf("    address %x, data %x, control %x, status %x\n",
         p->address, p->data, p->control, p->status);
        break;
      }
      case M_DATA: {
        printf("    block of %d bytes\n", bp->b_wptr - bp->b_rptr);
        break;
      }
    }
  }
}
#endif
```

6.0 IOC MULTIPLEXER DRIVER MODULE SOURCE LISTING

6.1 GLOBAL DECLARATIONS

```
#ifndef DEBUG
# define DEBUG 0
#endif

#define NIOC 65                    /* total number of channels plus one */

#include "epmsg.h"

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/errno.h>

#include <sun/vddrv.h>
#include <sys/conf.h>

static int iocopen();
static int iocclose();
static int iocuwput();
static int ioclwsvr();
static int ioclrput();

static struct module_info info = {
    0,                                  /* ID number */
    "ioc",                              /* name */
    0,                                  /* minimum packet size */
    INFPSZ,                             /* maximum packet size */
    512,                                /* flow control high water mark */
    128                                 /* flow control low water mark */
};

static struct qinit urinit = {
    NULL,                               /* put procedure */
    NULL,                               /* service procedure */
    iocopen,                            /* called on push or each open */
    iocclose,                           /* called on pop or last close */
    NULL,                               /* reserved for future use */
    &info,                              /* information structure */
    NULL                                /* statistics structure */
};
```

```
static struct qinit uwinit = {
    iocuwput,                       /* put procedure */
    NULL,                           /* service procedure */
    NULL,                           /* called on push or each open */
    NULL,                           /* called on pop or last close */
    NULL,                           /* reserved for future use */
    &info,                          /* information structure */
    NULL                            /* statistics structure */
};

static struct qinit lrinit = {
    ioclrput,                       /* put procedure */
    NULL,                           /* service procedure */
    NULL,                           /* called on push or each open */
    NULL,                           /* called on pop or last close */
    NULL,                           /* reserved for future use */
    &info,                          /* information structure */
    NULL                            /* statistics structure */
};

static struct qinit lwinit = {
    NULL,                           /* put procedure */
    ioclwsvr,                       /* service procedure */
    NULL,                           /* called on push or each open */
    NULL,                           /* called on pop or last close */
    NULL,                           /* reserved for future use */
    &info,                          /* information structure */
    NULL                            /* statistics structure */
};

static char *iocmodlist[] = {
    "ioa",
    NULL
};

struct streamtab iocinfo = {
    &urinit,                        /* upper read queue definitions */
    &uwinit,                        /* upper write queue definitions */
    &lrinit,                        /* lower read queue definitions */
    &lwinit,                        /* lower write queue definitions */
    iocmodlist                      /* list of modules to be pushed */
};

struct ioc {
    unsigned short io_reg[16];      /* IO device control registers */
    unsigned short io_stat[4];      /* IO channel status registers */
    unsigned short ei_word[2];      /* pending external interrupt word */
    unsigned short ei_size;         /* size of pending EI word in bytes */
```

```
    queue_t *qptr;                          /* pointer to read queue */
};
struct ioc ioc_ioc[NIOC];
int ioc_cnt = NIOC;

queue_t *iocbot;                            /* lower write queue */
int iocerr;

static queue_t *get_next_q();
static mblk_t *mk_rupt3();
static mblk_t *mk_eiw();

/* class 3 interrupt code to IOC status register 0 bit map */

static unsigned short rupt_to_stat[8] = {
    EI_RUPT_PEND,       /* 000 */
    0,                  /* 001 */
    IN_RUPT_PEND,       /* 010 */
    0,                  /* 011 */
    OUT_RUPT_PEND,      /* 100 */
    0,                  /* 101 */
    ICTO_RUPT_PEND,     /* 110 */
    0                   /* 111 */
};
```

6.2 LOADABLE MODULE STRUCTURES

```
extern int nulldev();
extern int nodev();

struct cdevsw ioccdev = {
    nodev, nodev, nodev, nodev, nodev, nodev,
    nodev, nodev, &iocinfo,
};

struct vdldrv ioc_vd = {
    VDMAGIC_PSEUDO,             /* Drv_magic */
    "ioc",                      /* Drv_name */
    NULL,                       /* Drv_mb_ctlr */
    NULL,                       /* Drv_mb_driver */
    NULL,                       /* Drv_numctlrs */
    0,                          /* Drv_numctlrs */
    NIOC,                       /* Drv_numdevs */
    0,                          /* Drv_bdevsw */
    &ioccdev,                   /* Drv_cdevsw */
    0,                          /* Drv_blockmajor */
    0                           /* Drv_charmajor */
};
```

6.3 LOADABLE MODULE INITIALIZATION

```
iocinit(function_code, vdp, vdi, vds)
unsigned int function_code;
struct vddrv *vdp;
caddr_t vdi;
struct vdstat *vds;
{
  switch(function_code) {

    case VDLOAD: {
      vdp->vdd_vdtab = (struct vdlinkage *)&ioc_vd;
    }
    case VDUNLOAD:
    case VDSTAT: {
      return(0);
    }

    default: {
      return(EIO);
    }
  }
}
```

6.4 OPEN ROUTINE

```
static int
iocopen(q, dev, flag, sflag)
queue_t *q;                     /* pointer to read queue */
dev_t dev;                      /* major and minor device numbers */
int flag;                       /* file open flags */
int sflag;                      /* stream open flags */
{
  struct ioc *ioc;

  if(sflag == CLONEOPEN) {
    for(dev = 0; dev < ioc_cnt; dev++) {
      if(ioc_ioc[dev].qptr == NULL) {
        break;
      }
    }
  }
  else {
    dev = minor(dev);
  }

  if(dev >= ioc_cnt) {
    return(OPENFAIL);
  }
```

```
  if(q->q_ptr) {                /* check to see if already open */
    u.u_error = EBUSY;
    return(OPENFAIL);
  }

  ioc = &ioc_ioc[dev];          /* pointer to minor device private structure
*/

  ioc->qptr = q;                /* pointer to read queue */
  q->q_ptr = (char *)ioc;       /* read queue link */
  WR(q)->q_ptr = (char *)ioc;   /* write queue link */

  ioc->io_stat[0] = ((dev - 1) & 0xF) + (ioc->io_stat[0] & ~0xF);

  return(dev);
}
```

6.5 UPPER WRITE PUT ROUTINE

```
static int
iocuwput(q, mp)
queue_t *q;
mblk_t *mp;
{
  struct ioc *ioc;

  ioc = (struct ioc *)q->q_ptr;

  switch(mp->b_datap->db_type) {
    case M_IOCTL: {
      struct iocblk *iocp;
      struct linkblk *linkp;

      iocp = (struct iocblk *)mp->b_rptr;

      switch(iocp->ioc_cmd) {
        case I_LINK: {
          if(ioc != ioc_ioc) {          /* nak if not device 0 */
            goto iocnak;
          }
          if(iocbot != NULL) {          /* nak if already linked */
            goto iocnak;
          }
          linkp = (struct linkblk *)mp->b_cont->b_rptr;
          iocbot = linkp->l_qbot;
          iocerr = 0;
          goto iocack;
        }
```

```
    case I_UNLINK: {
      if(ioc != ioc_ioc) {          /* nak if not device 0 */
        goto iocnak;
      }
      linkp = (struct linkblk *)mp->b_cont->b_rptr;
      iocbot = NULL;
    iocack:
      iocp->ioc_count = 0;
      mp->b_datap->db_type = M_IOCACK;
      break;
    }
    default: {
    iocnak:
      mp->b_datap->db_type = M_IOCNAK;
      break;
    }
  }
doreply:
  qreply(q, mp);
  break;
}

case M_FLUSH: {
  if(*mp->b_rptr & FLUSHW) {
    flushq(q, FLUSHDATA);
  }
  if(*mp->b_rptr & FLUSHR) {
    flushq(RD(q), FLUSHDATA);
    *mp->b_rptr &= ~FLUSHW;
    goto doreply;
  }
  freemsg(mp);
  break;
}

case M_PROTO:
case M_PCPROTO: {
  if(iocerr || iocbot == NULL) {
    goto bad;
  }
  putq(q, mp);                      /* put into upper write queue */
  qenable(iocbot);                  /* schedule lower write service */
  break;
}

default: {
bad:
  mp->b_datap->db_type = M_ERROR;
```

```
      mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
      *mp->b_wptr++ = EINVAL;
      goto doreply;
    }
  }
}
```

6.6 LOWER WRITE QUEUE SERVICE ROUTINE

```
static int
ioclwsvr(q)
register queue_t *q;                    /* lower write queue pointer */
{
  register mblk_t *mp;
  register queue_t *nq;                 /* upper write queue pointer */
  struct msg_cntl_part *p;
  struct ioc *iocdp;
  unsigned short *srp;                  /* status register pointer */

  while(canput(q->q_next)) {            /* check for room downstream */
    nq = get_next_q();                  /* find non-empty upper write queue */
    if(nq == NULL) {
      return;                           /* aren't any */
    }
    mp = getq(nq);                      /* get message from that queue */

    p = (struct msg_cntl_part *)(mp->b_datap->db_base);
    p->device = (struct ioc *)nq->q_ptr - ioc_ioc;
#if DEBUG > 1
prmsg(mp, "ioclwsvr");
#endif

    iocdp = &ioc_ioc[p->device];

    srp = &iocdp->io_stat[3];
    if(p->chain == OUTPUT && (*srp & OUT_CH_ABORT)) {
      goto discard;
    }
    else if(*srp & IN_CH_ABORT) {
      goto discard;
    }

    switch(p->function) {

      case READ_REG: {                  /* read device control register */
        p->data = iocdp->io_reg[p->control & 0xF];
      ackck:
        if(p->ack == DISCARD) {
```

```
discard:
    freemsg(mp);
    continue;
  }
  qreply(nq, mp);                    /* send back upstream */
  continue;
}

case WRITE_REG: {                    /* write device control register */
  int reg;
  reg = p->control & 0xF;
  iocdp->io_reg[reg] = p->data;
  if(reg == IN_CAP) {
    iocdp->io_stat[3] |= IN_CH_ACTIVE;
  }
  else if(reg == OUT_CAP) {
    iocdp->io_stat[3] |= OUT_CH_ACTIVE;
  }
  goto ackck;
}

case WRITE_STAT: {                   /* write channel status register */
  unsigned short *srp;
  srp = &iocdp->io_stat[p->address & 0x3];
  *srp = (p->data & p->control) + (*srp & ~p->control);
  goto ackck;
}

case READ_INST:                      /* get instruction address */
case READ_INST_Y: {                  /* get instruction y field address */
readinst:
  p->address = iocdp->io_reg[(p->chain == INPUT) ? IN_CAP : OUT_CAP]++;
  break;
}

case JUMP_STAT: {                    /* conditional jump instruction */
  if((iocdp->io_stat[0] & p->control) == 0) {
    goto readinst;
  }
}
case JUMP_INST: {                    /* unconditional jump instruction */
  iocdp->io_reg[(p->chain == INPUT) ? IN_CAP : OUT_CAP] = p->address +
1;
  break;
}

case REG_TO_MEM: {                   /* control register to memory */
  p->data = iocdp->io_reg[p->control & 0xF];
```

```
          break;
        }

        case STAT_TO_MEM: {                /* status register to memory */
          p->data = iocdp->io_stat[p->control & 0x3];
          break;
        }

        case WRITE_EI_WORD: {
          mblk_t *bp;
          struct msg_cntl_part *pp;

          if((bp = mp->b_cont) != NULL) {
            pp = (struct msg_cntl_part *)(bp->b_datap->db_base);
            pp->device = p->device;
          }
          if((iocdp->io_stat[3] & (EIE_LINE|RUPT_GEN)) == (EIE_LINE|RUPT_GEN)) {
            break;
          }
          bp = unlinkb(mp);
          if((iocdp->io_stat[3] & EIE_LINE) == 0) {
            iocdp->io_stat[0] |= EI_WORD_PEND;
            iocdp->ei_word[0] = p->address;
            iocdp->ei_word[1] = p->data;
            iocdp->ei_size = p->control;
#if DEBUG > 0
printf("ioclwsvr: external interrupt word pending\n");
#endif
            freeb(mp);
            mp = NULL;
          }
          if(bp != NULL) {
            if((iocdp->io_stat[3] & RUPT_GEN) == 0) {
              iocdp->io_stat[0] |= EI_RUPT_PEND;
#if DEBUG > 0
printf("ioclwsvr: external interrupt pending\n");
#endif
              freemsg(bp);
              bp = NULL;
            }
          }
          if(mp != NULL) {
            putnext(q, mp);
          }
          else if(bp != NULL) {
            putnext(q, bp);
          }
```

```
        continue;
      }

      case INTERRUPT: {
        if(p->data == 3) {              /* class 3 interrupt */
          if((iocdp->io_stat[3] & RUPT_GEN) == 0) {
            iocdp->io_stat[0] |= rupt_to_stat[p->address & 0x7];
#if DEBUG > 0
printf("ioclwsvr: interrupt pending\n");
#endif
            goto ackck;
          }
          p->address = (p->address & 0x7) + ((p->device - 1) << 3);
        }
        break;
      }

      case CHAN_CONTROL: {              /* channel control */
        int dev;
        unsigned short i;
        mblk_t *bp;

        bp = NULL;
        dev = ((p->device - 1) | 0xF) + 1;
        for(i = (1<<15); i != 0; i >>= 1) {
          if(dev < ioc_cnt && (p->address & i) != 0) {
            srp = &ioc_ioc[dev].io_stat[3];
            *srp |= p->data;
            *srp &= p->control;
            srp = &ioc_ioc[dev].io_stat[0];
            if(p->data & RUPT_GEN) {
              if(*srp & EI_RUPT_PEND) {
                *srp &= ~EI_RUPT_PEND;
                bp = mk_rupt3(bp, dev, 0x0);
              }
              if(*srp & OUT_RUPT_PEND) {
                *srp &= ~OUT_RUPT_PEND;
                bp = mk_rupt3(bp, dev, 0x4);
              }
              if(*srp & IN_RUPT_PEND) {
                *srp &= ~IN_RUPT_PEND;
                bp = mk_rupt3(bp, dev, 0x2);
              }
            }
            if(p->data & EIE_LINE) {
              if(*srp & EI_WORD_PEND) {
                *srp &= ~EI_WORD_PEND;
                bp = mk_eiw(bp, dev);
```

```
            }
          }
        }
        dev--;
      }
      if(bp != NULL) {
        putnext(q, bp);
      }
      goto ackck;
    }
  }
  putnext(q, mp);
 }
}
```

6.7 CLASS 3 INTERRUPT MESSAGE ROUTINE

```
static mblk_t *
mk_rupt3(bp, dev, code)
mblk_t *bp;
int dev;
int code;
{
  mblk_t *np;

  if((np = allocb(sizeof(struct msg_cntl_part), BPRI_MED)) == NULL) {
    printf("ioc: mk_rupt3: allocb failed\n");
  }
  else {
    np->b_datap->db_type = M_PROTO;
    *np->b_wptr++ = INTERRUPT;                  /* function field */
    *np->b_wptr++ = DISCARD;                    /* ack field */
    *np->b_wptr++ = (unsigned char)dev;         /* device field */
    *np->b_wptr++ = 0;                          /* chain field */
    *np->b_wptr++ = 0;                          /* address field */
    *np->b_wptr++ = ((dev - 1) << 3) + code;    /*  interrupt code */
    *np->b_wptr++ = 0;                          /* data field */
    *np->b_wptr++ = 3;                          /*  interrupt class 3 */
    *np->b_wptr++ = 0;                          /* control field */
    *np->b_wptr++ = 0;                          /*  status register 2 */
    *np->b_wptr++ = 0;                          /* status field */
    *np->b_wptr++ = 0;                          /*  not used */
    if(bp == NULL) {
      return(np);
    }
    linkb(bp, np);
  }
```

```
  return(bp);
}
```

6.8 EXTERNAL INTERRUPT WORD MESSAGE ROUTINE

```
static mblk_t *
mk_eiw(bp, dev)
mblk_t *bp;
int dev;
{
  mblk_t *np;
  struct ioc *p;

  p = &ioc_ioc[dev];

  if((np = allocb(sizeof(struct msg_cntl_part), BPRI_MED)) == NULL) {
    printf("ioc: mk_eiw: allocb failed\n");
  }
  else {
    np->b_datap->db_type = M_PROTO;
    *np->b_wptr++ = WRITE_EI_WORD;              /* function field */
    *np->b_wptr++ = DISCARD;                    /* ack field */
    *np->b_wptr++ = (unsigned char)dev;         /* device field */
    *np->b_wptr++ = 0;                          /* chain field */
    *np->b_wptr++ = p->ei_word[0] >> 8;         /* address field */
    *np->b_wptr++ = p->ei_word[0];              /*  32-bit EI word extension
*/
    *np->b_wptr++ = p->ei_word[1] >> 8;         /* data field */
    *np->b_wptr++ = p->ei_word[1];              /*  EI word data */
    *np->b_wptr++ = 0;                          /* control field */
    *np->b_wptr++ = p->ei_size;                 /*  EI word size in bytes */
    *np->b_wptr++ = 0;                          /* status field */
    *np->b_wptr++ = 0;                          /*  not used */
    if(bp == NULL) {
      return(np);
    }
    linkb(bp, np);
  }
  return(bp);
}
```

6.9 ROUND-ROBIN UPPER WRITE QUEUE SCHEDULER

```
static queue_t *
get_next_q()
{
```

```
    static int next;                        /* next device to be checked */
    int i;
    int start;                              /* first device to be checked */
    register queue_t *q;

    start = next;
    for(i = next; i < ioc_cnt; i++) {
      if(q = ioc_ioc[i].qptr) {
        q = WR(q);
        if(q->q_first != NULL) {            /* if queue non-empty */
          next = i + 1;
          return(q);
        }
      }
    }
    for(i = 0; i < start; i++) {
      if(q = ioc_ioc[i].qptr) {
        q = WR(q);
        if(q->q_first != NULL) {            /* if queue non-empty */
          next = i + 1;
          return(q);
        }
      }
    }
    return(NULL);                           /* all queues empty */
}
```

6.10 LOWER READ PUT ROUTINE

```
static int
ioclrput(q, mp)
queue_t *q;
mblk_t *mp;
{
  queue_t *uq;
  int dev;

  switch(mp->b_datap->db_type) {

    case M_FLUSH: {
      if(*mp->b_rptr & FLUSHR) {
        flushq(q, 0);
      }
      if(*mp->b_rptr & FLUSHW) {
        *mp->b_rptr &= ~FLUSHR;
        qreply(q, mp);
      }
```

```
    else {
      freemsg(mp);
    }
    break;
  }

  case M_ERROR:
  case M_HANGUP: {
    iocerr = 1;
    freemsg(mp);
    break;
  }

  case M_PROTO:
  case M_PCPROTO: {
    struct msg_cntl_part *p;
    struct ioc *iocdp;
                                        /* retrieve device number */
    p = (struct msg_cntl_part *)(mp->b_datap->db_base);
    dev = p->device;

    if(dev < 0 || dev >= ioc_cnt) {    /* check device number range */
      goto discard;
    }

    uq = ioc_ioc[dev].qptr;            /* identify upstream read queue */
    if(uq == NULL) {
      goto discard;                    /* discard message if no queue */
    }

    iocdp = &ioc_ioc[dev];

    switch(p->function) {

      case MEM_TO_REG: {
        iocdp->io_reg[p->control & 0xF] = p->data;
        break;
      }

      case READ_START: {
      rstart:
        iocdp->io_stat[0] |= OUT_BUF_ACTIVE;
        goto readbcw;
      }
      case READ_BUF: {
      rbuf:
        iocdp->io_stat[0] &= ~OUT_BUF_ACTIVE;
      readbcw:
        iocdp->io_reg[OUT_BCW] = p->control;
```

```
    iocdp->io_reg[OUT_BAP] = p->address;
    break;
  }

  case WRITE_START: {
  wstart:
    iocdp->io_stat[0] |= IN_BUF_ACTIVE;
    goto writebcw;
  }
  case WRITE_BUF: {
  wbuf:
    iocdp->io_stat[0] &= ~IN_BUF_ACTIVE;
  writebcw:
    iocdp->io_reg[IN_BCW] = p->control;
    iocdp->io_reg[IN_BAP] = p->address;
    break;
  }

  case EF_START: {
    if(p->chain == OUTPUT) {
      goto rstart;
    }
    goto wstart;
  }

  case EF_BUF: {
    if(p->chain == OUTPUT) {
      goto rbuf;
    }
    goto wbuf;
  }

  case TEST_AND_SET: {
    if(p->data & BF_FLAG) {
      goto setcond;
    }
    else {
      goto clearcond;
    }
  }

  case TEST_WORD: {
    if(p->data & p->control) {
  clearcond:
      iocdp->io_stat[0] &= ~COND_TEST;
      break;
    }
    else {
```

```
        setcond:
            iocdp->io_stat[0] |= COND_TEST;
            break;
          }
        }

        case INTERRUPT: {
          if(p->data == 3) {            /* class 3 interrupt */
            iocdp->io_stat[0] &= ~rupt_to_stat[p->address & 0x7];
          }
          break;
        }
      }

      if((p->ack != DISCARD) && canput(uq->q_next)) {
        putnext(uq, mp);                /* send upstream if possible */
        break;
      }
      else {
        goto discard;                   /* otherwise discard */
      }
    }    default: {
    discard:
      freemsg(mp);
    }
  }
}
```

6.11 CLOSE ROUTINE

```
static int
iocclose(q)
queue_t *q;                             /* pointer to read queue */
{

  ((struct ioc *)q->q_ptr)->qptr = NULL;
}
```

6.12 PRINT MESSAGE DEBUG ROUTINE

```
#if DEBUG > 0
static char *function_str[] = {
    "NO_FUNCTION",      /*  0   no function specified          */
    "IO_CELL",          /*  1   IO command cell                */
    "READ_INST",        /*  2   read instruction               */
    "READ_INST_Y",      /*  3   read instruction y field       */
```

```
    "JUMP_INST",        /*  4   jump to new instruction          */
    "JUMP_STAT",        /*  5   jump if status bit set           */
    "READ_REG",         /*  6   read control register            */
    "WRITE_REG",        /*  7   write control register           */
    "REG_TO_MEM",       /*  8   copy control register to memory  */
    "MEM_TO_REG",       /*  9   copy memory to control register  */
    "READ_START",       /* 10   read BCW and BAP                 */
    "READ_BUF",         /* 11   read memory user data buffer     */
    "WRITE_START",      /* 12   write BCW and BAP                */
    "WRITE_BUF",        /* 13   write memory user data buffer    */
    "EF_START",         /* 14   read BCW and BAP                 */
    "EF_BUF",           /* 15   read memory user data buffer     */
    "WRITE_WORD",       /* 16   atomic read-modify-write         */
    "INTERRUPT",        /* 17   generate UYK-44 interrupt        */
    "STAT_TO_MEM",      /* 18   copy status register to memory   */
    "WRITE_STAT",       /* 19   write status register            */
    "TEST_WORD",        /* 20   test memory word                 */
    "TEST_AND_SET",     /* 21   test and set memory word         */
    "CHAN_CONTROL",     /* 22   channel control                  */
    "WRITE_EI_WORD"     /* 23   write external interrupt word    */
};

static int
prmsg(mp, s)
mblk_t *mp;
char *s;
{
  struct msg_cntl_part *p;
  mblk_t *bp;
  unsigned short t;  if(s) {
    printf("%s:\n", s);
  }

  for(bp = mp; bp != NULL; bp = bp->b_cont) {
    switch(bp->b_datap->db_type) {
      case M_PROTO:
      case M_PCPROTO: {
        p = (struct msg_cntl_part *)(bp->b_datap->db_base);
        t = p->function;
        if(t > 0 && t < (sizeof(function_str) / sizeof(*function_str))) {
          printf("  %s", function_str[t]);
        }
        else {
          printf("  %x", t);
        }
        printf(", ack %s", (p->ack == DISCARD) ? "DISCARD" : "REPLY");
        printf(", device %x", p->device);
        printf(", chain %s\n", (p->chain == OUTPUT) ? "OUTPUT" : "INPUT");
```

```
        printf("    address %x, data %x, control %x, status %x\n",
         p->address, p->data, p->control, p->status);
        break;
      }
      case M_DATA: {
        printf("    block of %d bytes\n", bp->b_wptr - bp->b_rptr);
        break;
      }
    }
  }
}
#endif
```

6.13 PRINT CHANNEL REGISTER DEBUG ROUTINE

```
#if DEBUG > 0
static int
prreg(mp, s)
mblk_t *mp;
char *s;
{
  struct msg_cntl_part *p;
  int i;  if(s) {
    printf("%s:\n", s);
  }

  p = (struct msg_cntl_part *)(mp->b_datap->db_base);

  printf("  io_reg[0-7]:");
  for(i = 0; i < 8; i++) {
    printf(" %x", ioc_ioc[p->device].io_reg[i]);
  }
  printf("\n  io_stat[0-3]:");
  for(i = 0; i < 4; i++) {
    printf(" %x", ioc_ioc[p->device].io_stat[i]);
  }
  printf("\n");
}
#endif
```

7.0 IOA MODULE SOURCE LISTING

7.1 GLOBAL DECLARATIONS

```
#ifndef DEBUG
# define DEBUG 0
#endif

#include "epmsg.h"
#include "epcntl.h"

#include <sys/types.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/dir.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/errno.h>
#include <sun/vddrv.h>
#include <sys/conf.h>

/* stream related declarations */

static int ioaopen();   /* open routine, called on each open */
static int ioarput();   /* read QUEUE put procedure */
static int ioarsvr();   /* read QUEUE service routine */
static int ioawput();   /* write QUEUE put procedure */
static int ioawsvr();   /* write QUEUE service routine */
static int ioaclose();  /* close routine, called on last close */

static struct module_info minfo = {
    0,                  /* ID number */
    "ioa",              /* name */
    0,                  /* QUEUE minimum packet size accepted */
    INFPSZ,             /* QUEUE maximum packet size accepted */
    150,                /* QUEUE flow control high water mark */
    50                  /* QUEUE flow control low water mark */
};

static struct qinit rinit = {
    ioarput,            /* read QUEUE put routine */
    ioarsvr,            /* read QUEUE service routine */
    ioaopen,            /* open routine, called on each open */
    ioaclose,           /* close routine, called on last close */
```

```
    NULL,               /* reserved */
    &minfo,             /* pointer to read QUEUE information structure */
    NULL                /* pointer to read QUEUE statistics structure */
};

static struct qinit winit = {
    ioawput,            /* write QUEUE put routine */
    ioawsvr,            /* write QUEUE service routine */
    NULL,               /* ignored */
    NULL,               /* ignored */
    NULL,               /* reserved */
    &minfo,             /* pointer to write QUEUE information structure */
    NULL                /* pointer to write QUEUE statistics structure */
};

struct streamtab ioainfo = {
    &rinit,             /* pointer to read QUEUE initialization structure */
    &winit,             /* pointer to write QUEUE initialization structure */
    NULL,               /* multiplexer drivers only - not used */
    NULL,               /* multiplexer drivers only - not used */
    NULL                /* pointer to module push list on first open */
};
```

7.2 LOADABLE MODULE STRUCTURES

```
extern struct fmodsw fmodsw[];

static struct fmodsw ioa_fsw = {
    "ioa",
    &ioainfo,
};

static int ioa_loaded;static struct vdldrv ioa_drv = {
    VDMAGIC_PSEUDO,     /* Drv_magic */
    "ioa",              /* Drv_name */
#if sun4c
    NULL,               /* Drv_dev_ops */
#else
    NULL,               /* Drv_mb_ctlr */
    NULL,               /* Drv_mb_driver */
    NULL,               /* Drv_mb_device */
    0,                  /* Drv_numctlrs */
    0,                  /* Drv_numdevs */
#endif
    0,                  /* Drv_bdevsw */
    0,                  /* Drv_cdevsw */
    0,                  /* Drv_blockmajor */
```

```
    0                   /* Drv_charmajor */
};
```

7.3 LOADABLE MODULE INITIALIZATION

```
int
ioainit(command, vdp, vdi, vds)
int command;
struct vddrv *vdp;
struct vdioctl_load *vdi;
struct vdstat *vds;
{
  extern int fmodcnt;
  int i;

  switch(command) {

    case VDLOAD: {

      /* Since loadable STREAMS modules don't work, link ourselves */
      /* into the fmodsw. */

      /* find a free slot */
      i = 0;
      while((fmodsw[i].f_name[i] == '\0') && (++i < fmodcnt)) /* null */ ;
      if(i >= fmodcnt) {
        return(ENOSPC);
      }
      fmodsw[i] = ioa_fsw;
      vdp->vdd_vdtab = (struct vdlinkage *)&ic _drv;
      return(0);
    }

    case VDUNLOAD: {
      if(ioa_loaded) {
        return(EBUSY);
      }

      i = 0;
      /* find the loaded version. Don't look for "ioa" */
      /* in case it is configured in too. */
      while(fmodsw[i].f_str != ioa_fsw.f_str) {
        i++;
      }

      /* now move everyone down one level */
      do {
        fmodsw[i] = fmodsw[i+1];
```

```
      } while((fmodsw[i+1].f_name[0] != '\0') && (++i<(fmodcnt-1)));
      return(0);
    }

    case VDSTAT: {
      return(0);
    }

    default: {
      return(EINVAL);
    }
  }
}
```

7.4 OPEN ROUTINE

```
static int
ioaopen(q, dev, flag, sflag)
queue_t *q;                     /* pointer to read QUEUE */
dev_t dev;                      /* major and minor device, 0 for modules */
int flag;                       /* file open flags, 0 for modules */
int sflag;                      /* stream open flags */
{
#if DEBUG > 0
printf("ioa open, dev %x\n", dev & 0xFF);
#endif
  return(0);
}
```

7.5 READ PUT ROUTINE

```
static int
ioarput(q, mp)
queue_t *q;                     /* pointer to read queue */
mblk_t *mp;                     /* pointer to message */
{

  switch(mp->b_datap->db_type) {

    case M_FLUSH: {
      if(*mp->b_rptr & FLUSHR) {        /* if flush read flag set */
        flushq(q, FLUSHDATA);           /* flush read QUEUE */
      }
    }
    default: {
      putnext(q, mp);                   /* send message upstream */
      break;
    }
```

```
    case M_PROTO:
    case M_PCPROTO: {
      putq(q, mp);                          /* put message on read queue */
      qenable(q);
      break;
    }
  }
  return(0);
}
```

7.6 READ QUEUE SERVICE ROUTINE

```
static int
ioarsvr(q)
queue_t *q;
{
  mblk_t *mp;
  struct msg_cntl_part *p;

  while(mp = getq(q)) {                     /* only M_PROTO messaged were queued
*/

    p = (struct msg_cntl_part *)(mp->b_datap->db_base);
#if DEBUG > 0
prmsg(mp, "ioarsvr");
#endif

    if(p->status) {                         /* if downstream IO fault */
      mp->b_datap->db_type = M_ERROR;
      mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
      *mp->b_wptr++ = EIO;                  /* send IO error upstream */
      putnext(q, mp);
    }

    switch(p->function) {

      case IO_CELL: {
        switch(p->control >> 8) {           /* switch on opcode */
          case 0xE0: {                      /* RR channel control */
            p->function = CHAN_CONTROL;
            switch(p->control & 0xF) {      /* switch on m field */
              case 0x0: {                   /* clear all channels */
                p->data = p->control = (IN_CH_ABORT|OUT_CH_ABORT);
              allchan:
                p->address = 0xFFFF;
                goto cmdmsg;
              }
              case 0x4: {                   /* enable all channel EI data */
```

```
    p->data = EIE_LINE;
    p->control = 0xFFFF;
    goto allchan;
  }
  case 0x5: {                   /* disable all channel EI data */
    p->data = 0;
    p->control = ~EIE_LINE;
    goto allchan;
  }
  case 0x6: {                   /* enable lower channel interrupts */
    p->address = (p->control & 0x00F0)
                 ? (1 << ((p->control >> 4) & 0xF)) - 1
                 : 0xFFFF;
    p->data = RUPT_GEN;
    p->control = 0xFFFF;
    goto cmdmsg;
  }
  case 0x7: {                   /* disable lower channel interrupts */
    p->address = (p->control & 0x00F0)
                 ? (1 << ((p->control >> 4) & 0xF)) - 1
                 : 0xFFFF;
    p->data = 0;
    p->control = ~RUPT_GEN;
    goto cmdmsg;
  }
  case 0x8: {                   /* clear channel a */
    p->data = (IN_CH_ABORT|OUT_CH_ABORT);
  cleara:
    p->control = (IN_CH_ABORT|OUT_CH_ABORT);
  chana:
    p->address = 1 << ((p->control >> 4) & 0xF);
    goto cmdmsg;
  }
  case 0x9: {                   /* clear channel a input */
    p->data = IN_CH_ABORT;
    goto cleara;
  }
  case 0xA: {                   /* clear channel a output */
    p->data = OUT_CH_ABORT;
    goto cleara;
  }
  case 0xC: {                   /* enable channel a EI data */
    p->data = EIE_LINE;
    p->control = 0xFFFF;
    goto chana;
  }
  case 0xD: {                   /* disable channel a EI data */
```

```
            p->data = 0;
            p->control = -EIE_LINE;
            goto chana;
          }
          case 0xE: {                  /* enable channel a interrupts */
            p->data = RUPT_GEN;
            p->control = 0xFFFF;
            goto chana;
          }
          case 0xF: {                  /* disable channel a interrupts */
            p->data = 0;
            p->control = -RUPT_GEN;
            goto chana;
          }
          default: {
            p->control = 3;
            goto badcmd;
          }
        }
      }
      case 0xE6: {
        p->function = WRITE_REG;
        p->data = p->address;
        switch(p->control & 0xF) {
          case 2: {                    /* initiate input chain */
            p->chain = INPUT;
            p->ack = REPLY;
            goto replymsg;
          }
          case 6: {                    /* initiate output chain */
            p->chain = OUTPUT;
            p->ack = REPLY;
            goto replymsg;
          }
          default: {                   /* RK load channel control register */
            goto cmdmsg;
          }
        }
      }
      case 0xE7: {                     /* RX load channel control register */
        p->function = MEM_TO_REG;
        goto cmdmsg;
      }
      case 0xEB: {                     /* RX store channel control register
*/
        p->function = REG_TO_MEM;
        goto cmdmsg;
```

```
        }
        case 0xF8: {                    /* set/clear discretes */
          /* not coded yet */
          goto discardmsg;
        }
        case 0xFB: {                    /* RX store status register */
          if((p->control & 0xE) == 0x8) {
            p->function = STAT_TO_MEM;
            p->control = 0;
            goto cmdmsg;
          }
        }
        default: {                      /* Illegal IOC command */
          p->control = 3;
          goto badcmd;
        }
      }
    }

    case READ_INST:
    case JUMP_INST:
    case JUMP_STAT: {
      switch(p->data >> 8) {
        case 0xE3:                      /* RX initiate transfer */
        case 0xE6:                      /* RK load channel control .egister */
        case 0xE7:                      /* RX load channel control register */
        case 0xEB:                      /* RX store channel control register
*/
        case 0xEF:                      /* RX set/clear flag */
        case 0xF2:                      /* RK conditional jump */
        case 0xFB: {                    /* RX store channel status register */
          p->function = READ_INST_Y;
          p->control = p->data;
          goto replymsg;
        }
        case 0xE0: {                    /* RR channel control */
          p->function = CHAN_CONTROL;
          switch(p->data & 0xF) {       /* switch on m field */
            case 0x0: {                 /* clear all channels */
              p->data = p->control = (IN_CH_ABORT|OUT_CH_ABORT);
            allchanx:
              p->address = 0xFFFF;
              goto replymsg;
            }
            case 0x4: {                 /* enable all channel EI data */
              p->data = EIE_LINE;
              p->control = 0xFFFF;
```

```
  goto allchanx;
}
case 0x5: {                /* disable all channel EI data */
  p->data = 0;
  p->control = ~EIE_LINE;
  goto allchanx;
}
case 0x6: {                /* enable lower channel interrupts */
  p->address = (p->control & 0x00F0)
             ? (1 << ((p->control >> 4) & 0xF)) - 1
             : 0xFFFF;
  p->data = RUPT_GEN;
  p->control = 0xFFFF;
  goto cmdmsg;
}
case 0x7: {                /* disable lower channel interrupts */
  p->address = (p->control & 0x00F0)
             ? (1 << ((p->control >> 4) & 0xF)) - 1
             : 0xFFFF;
  p->data = 0;
  p->control = ~RUPT_GEN;
  goto cmdmsg;
}
case 0x8: {                /* clear channel a */
  p->data = (IN_CH_ABORT|OUT_CH_ABORT);
clearx:
  p->control = (IN_CH_ABORT|OUT_CH_ABORT);
chandev:
  p->address = 1 << ((p->device - 1) & 0xF);
  goto replymsg;
}
case 0x9: {                /* clear channel a input */
  p->data = IN_CH_ABORT;
  goto clearx;
}
case 0xA: {                /* clear channel a output */
  p->data = OUT_CH_ABORT;
  goto clearx;
}
case 0xC: {                /* enable channel a EI data */
  p->data = EIE_LINE;
  p->control = 0xFFFF;
  goto chandev;
}
case 0xD: {                /* disable channel a EI data */
  p->data = 0;
  p->control = ~EIE_LINE;
```

```
            goto chandev;
          }
          case 0xE: {                  /* enable channel a interrupts */
            p->data = RUPT_GEN;
            p->control = 0xFFFF;
            goto chandev;
          }
          case 0xF: {                  /* disable channel a interrupts */
            p->data = 0;
            p->control = ~RUPT_GEN;
            goto chandev;
          }
          default: {
            goto badinst;
          }
        }
      }
      case 0xEC: {
        if(p->data & 0x0010) {         /* RR interrupt processor */
          p->data = CLASS_3;
          p->address = (p->chain == INPUT) ? 0x2 : 0x4;
          p->control = 0;
          goto dorupt;
        }
        else {                         /* RR halt chain */
          p->address = 1 << ((p->device - 1) & 0xF);
          p->data = 0;
          p->control = (p->chain == INPUT)
                     ? ~IN_CH_ACTIVE : ~OUT_CH_ACTIVE;
          p->function = CHAN_CONTROL;
          goto cmdmsg;
        }
      }
      default: {                       /* Illegal chain instruction */
      badinst:
        p->control = (p->chain == INPUT) ? 0 : 1;
      badcmd:
        p->ack = DISCARD;
        p->data = CLASS_2;
        p->address = COMMAND_FAULT;
        p->control += (p->device - 1) << 2;
      dorupt:
        p->function = INTERRUPT;
        goto replymsg;
      }
    }
  }
```

```
case READ_INST_Y: {
  switch(p->control >> 8) {
    case 0xE3: {                    /* RX initiate transfer */
      switch((p->control >> 4) & 0x3) {
        case 0: {                   /* Input to UYK-44 */
          p->function = WRITE_START;
          goto replymsg;
        }
        case 1: {                   /* Output from UYK-44 */
          p->function = READ_START;
          goto replymsg;
        }
        case 3: {                   /* Output forced external function */
          mp->b_datap->db_type = M_PCPROTO;
        }
        case 2: {                   /* Output external function */
          p->function = EF_START;
          goto replymsg;
        }
        default: {
          goto badinst;
        }
      }
    }
    case 0xE6: {                    /* RK load control memory */
      p->function = WRITE_REG;
      goto replymsg;
    }
    case 0xE7: {                    /* RX load control memory */
      p->function = MEM_TO_REG;
      p->address = p->data;
      goto replymsg;
    }
    case 0xEB: {                    /* RX store control register */
      p->function = REG_TO_MEM;
      p->address = p->data;
      goto replymsg;
    }
    case 0xEF: {                    /* RX set/clear flag */
      p->address = p->data;
      switch((p->control >> 4) & 0xF) {
        case 0: {                   /* RX zero flag */
          p->function = WRITE_WORD;
          p->data = ALL_ZEROS;
          p->control = BF_FLAG;
          goto replymsg;
        }
```

```
      case 1: {                    /* RX set flag */
        p->function = WRITE_WORD;
        goto setflag;
      }
      case 2: {                    /* RX test and set flag */
        p->function = TEST_AND_SET;
      setflag:
        p->data = ALL_ONES;
        p->control = BF_FLAG;
        goto replymsg;
      }
      case 4: {                    /* RX zero bit m at Y */
        p->function = WRITE_WORD;
        p->data = ALL_ZEROS;
        goto mbit;
      }
      case 5: {                    /* RX set bit m at Y */
        p->function = WRITE_WORD;
        goto setmbit;
      }
      case 7: {                    /* RX compare bit m at Y to zero */
        p->function = TEST_WORD;
      setmbit:
        p->data = ALL_ONES;
      mbit:
        p->control = (1 << (p->control & 0xF));
        goto replymsg;
      }
      default: {                   /* Illegal chain instruction */
        goto badinst;
      }
    }
  }
  case 0xF2: {                     /* RK conditional jump */
    p->address = p->data;
    switch((p->control >> 4) & 0xF) {
      case 0: {                    /* unconditional jump */
        p->function = JUMP_INST;
        goto replymsg;
      }
      case 4: {                    /* jump if status COND_TEST set */
        p->function = JUMP_STAT;
        p->control = COND_TEST;
        goto replymsg;
        break;
      }
      case 8: {                    /* jump if status IN_BUF_ACTIVE set */
```

```
                p->function = JUMP_STAT;
                p->control = IN_BUF_ACTIVE;
                goto replymsg;
              }
              case 9: {                    /* jump if status OUT_BUF_ACTIVE set
*/
                p->function = JUMP_STAT;
                p->control = OUT_BUF_ACTIVE;
                goto replymsg;
              }
              defualt: {
                goto badinst;
              }
            }
          }
          case 0xFB: {                     /* RX store status register */
            if((p->control & 0xE) == 0x8) {
              p->function = STAT_TO_MEM;
              p->address = p->data;
              p->control = 0;
              goto replymsg;
            }
            goto badinst;
          }
        }
        break;
      }

      case WRITE_START: {
        p->function = WRITE_BUF;
        p->chain = INPUT;                  /* converts output chains to input */
                                           /* overcome write queue blocking */
        mp->b_datap->db_type = M_PCPROTO;
        putq(WR(q), mp);                   /* put message on write queue */
        return(0);
      }

      case READ_START: {
        p->function = READ_BUF;
        p->chain = OUTPUT;                 /* converts input chains to output */
        goto replymsg;
      }

      case EF_START: {
        p->function = EF_BUF;
        p->chain = OUTPUT;
        goto replymsg;
      }
```

```
      case EF_BUF:
      case READ_BUF: {
        mblk_t *np;

        np = unlinkb(mp);               /* disconnect M_PROTO header */
        if(np != NULL) {
          if(p->function == EF_BUF) {
            np->b_datap->db_type = mp->b_datap->db_type;
            mp->b_datap->db_type = M_PROTO;
          }
          putnext(q, np);               /* send data upstream */
        }
      }
      default: {
        p->function = READ_INST;
        p->ack = REPLY;
        goto replymsg;
      }
    }
  discardmsg:
    freemsg(mp);
    continue;  cmdmsg:
    p->ack = DISCARD;
#if DEBUG > 0
prmsg(mp, "cmdmsg");
#endif
  replymsg:
    if(p->limit != 0) {
      p->limit--;
      if(p->limit == 0) {
        (void)printf("limited terminated chain\n");
        goto discardmsg;
      }
    }
    qreply(q, mp);
    continue;
  }
}
```

7.7 WRITE PUT ROUTINE

```
static int
ioawput(q, mp)
queue_t *q;                     /* pointer to write QUEUE */
mblk_t *mp;                     /* pointer to message */
{
```

```
  switch(mp->b_datap->db_type) {

    case M_FLUSH: {
      if(*mp->b_rptr & FLUSHW) {          /* if flush write flag set */
        flushq(q, FLUSHDATA);             /* flush write QUEUE */
      }
    }
    default: {
    tonext:
      putnext(q, mp);                     /* send message downstream */
      break;
    }

    case M_DATA: {
      putq(q, mp);                        /* put data into write queue */
      qenable(q);
      break;
    }

    case M_IOCTL: {
      struct iocblk *iocp;
      mblk_t *np;
      mblk_t *bp;

      iocp = (struct iocblk *)mp->b_rptr;

      if(iocp->ioc_cmd == EXT_INTR) {
        unsigned short eichar[4];

        switch(iocp->ioc_count) {
          case 4: {
            eichar[0] = *mp->b_cont->b_rptr++;
            eichar[1] = *mp->b_cont->b_rptr++;
          }
          case 2: {
            eichar[2] = *mp->b_cont->b_rptr++;
          }
          case 1: {
            eichar[3] = *mp->b_cont->b_rptr++;

            if((np = allocb(sizeof(struct msg_cntl_part), BPRI_MED)) == NULL)
{
              goto badalloc;
            }
            if((bp = allocb(sizeof(struct msg_cntl_part), BPRI_MED)) == NULL)
{
              freeb(np);
            badalloc:
              printf("ioa: ioawput: allocb failed\n");
```

```
        goto badcntl;
      }

      np->b_datap->db_type = M_PROTO;
      *np->b_wptr++ = WRITE_EI_WORD;        /* function field */
      *np->b_wptr++ = DISCARD;              /* ack field */
      *np->b_wptr++ = 0;                    /* device field */
      *np->b_wptr++ = 0;                    /* chain field */
      *np->b_wptr++ = eichar[0];            /* address field */
      *np->b_wptr++ = eichar[1];            /*  EI word extension */
      *np->b_wptr++ = eichar[2];            /* data field */
      *np->b_wptr++ = eichar[3];            /*  EI word data */
      *np->b_wptr++ = 0;                    /* control field */
      *np->b_wptr++ = iocp->ioc_count;      /*  EI word size in bytes */
      *np->b_wptr++ = 0;                    /* status field */
      *np->b_wptr++ = 0;                    /*  not used */

      bp->b_datap->db_type = M_PROTO;
      *bp->b_wptr++ = INTERRUPT;            /* function field */
      *bp->b_wptr++ = DISCARD;              /* ack field */
      *bp->b_wptr++ = 0;                    /* device field */
      *bp->b_wptr++ = 0;                    /* chain field */
      *bp->b_wptr++ = 0;                    /* address field */
      *bp->b_wptr++ = 0;                    /*  interrupt code */
      *bp->b_wptr++ = 0;                    /* data field */
      *bp->b_wptr++ = 3;                    /*  interrupt class 3 */
      *bp->b_wptr++ = 0;                    /* control field */
      *bp->b_wptr++ = 0;                    /*  status register 2 */
      *bp->b_wptr++ = 0;                    /* status field */
      *bp->b_wptr++ = 0;                    /*  not used */

      linkb(np, bp);                        /* append interrupt */
      putnext(q, np);                       /* send downstream */

      iocp->ioc_error = 0;
      iocp->ioc_rval = 0;
      mp->b_datap->db_type = M_IOCACK;
      break;
    }
    default: {
    badcntl:
      iocp->ioc_error = EINVAL;
      iocp->ioc_rval = -1;
      mp->b_datap->db_type = M_IOCNAK;
      break;
    }
  }
  if(mp->b_cont) {
```

```
            freemsg(mp->b_cont);
            mp->b_cont = NULL;
          }
        ctlrpy:
          iocp->ioc_count = 0;
          qreply(q, mp);
          break;
        }
        else if(iocp->ioc_cmd == CHANNEL_TYPE) {
          unsigned char chtype;

          if(iocp->ioc_count != 1) {
            goto badcntl;
          }

          chtype = *mp->b_cont->b_rptr++;

          if((np = allocb(sizeof(struct msg_cntl_part), BPRI_MED)) == NULL) {
            goto badalloc;
          }
          np->b_datap->db_type = M_PROTO;
          *np->b_wptr++ = WRITE_STAT;              /* function field */
          *np->b_wptr++ = DISCARD;                 /* ack field */
          *np->b_wptr++ = 0;                       /* device field */
          *np->b_wptr++ = 0;                       /* chain field */
          *np->b_wptr++ = 0;                       /* address field */
          *np->b_wptr++ = 0;                       /*  status register 0 */
          *np->b_wptr++ = 0;                       /* data field */
          *np->b_wptr++ = chtype << 4;             /*  channel type byte */
          *np->b_wptr++ = 0;                       /* control field */
          *np->b_wptr++ = 0xF0;                    /*  channel type field mask */
          *np->b_wptr++ = 0;                       /* status field */
          *np->b_wptr++ = 0;                       /*  not used */
          putq(q, np);                             /* queue downstream */

          iocp->ioc_error = 0;
          iocp->ioc_rval = 0;
          mp->b_datap->db_type = M_IOCACK;
          goto ctlrpy;
        }
        goto tonext;
      }
    }
    return(0);
}
```

7.8 WRITE QUEUE SERVICE ROUTINE

```
static int
ioawsvr(q)
queue_t *q;                     /* pointer to write QUEUE */
{
  mblk_t *pmp;                  /* pointer to M_PROTO WRITE_BUF message */
  mblk_t *dmp;                  /* pointer to M_DATA message */
  mblk_t *bp;                   /* pointer to allocated message block */
  struct msg_cntl_part *p;      /* pointer to M_PROTO data block */
  unsigned short count;         /* number of bytes desired by UYK-44 */
  int size;                     /* number of bytes in M_DATA data block */
  int getcount;                 /* number of getq's required */

#if DEBUG > 1
printf("ioawsvr:\n");
#endif
  pmp = q->q_first;
  if(pmp == NULL) {             /* check for empty queue */
#if DEBUG > 1
printf("  Empty queue\n");
#endif
    return;
  }

  if(pmp->b_datap->db_type != M_PCPROTO) {
#if DEBUG > 1
printf("  No available first M_PCPROTO block\n");
#endif
    return;
  }

  if(pmp->b_next == NULL) {
#if DEBUG > 1
printf("  No available Unix M_DATA blocks\n");
#endif
    return;
  }

  p = (struct msg_cntl_part *)(pmp->b_datap->db_base);
  count = p->control & 0xFFF;
  if(count == 0) {
    count = 0x1000;
  }
  if(p->control & 0x8000) {
    count <<= 1;
    if(p->control & 0x4000) {
      count <<= 1;
```

```
    }
  }
#if DEBUG > 1
printf("  UYK-44 chain M_PCPROTO requires %d bytes\n", count);
#endif

  for(dmp = pmp->b_cont; dmp != NULL; dmp = dmp->b_cont) {
    count -= (dmp->b_wptr - dmp->b_rptr);
  }
#if DEBUG > 1
printf("  UYK-44 chain M_PCPROTO still needs %d bytes\n", count);
#endif

  getcount = 0;

  for(dmp = pmp->b_next; dmp != NULL; dmp = dmp->b_next) {

    if(dmp->b_datap->db_type == M_DATA) {
      size = dmp->b_wptr - dmp->b_rptr;
#if DEBUG > 1
printf("  Unix M_DATA has %d bytes\n", size);
#endif

      if(size == 0) {
        getcount++;
      }
      else if(size <= count) {
        if((bp = allocb(size, BPRI_MED)) == NULL) {
          printf("ioawsvr: size alloc failed\n");
        }
        else {
          getcount++;
          count -= size;
          while(size--) {
            *bp->b_wptr++ = *dmp->b_rptr++;
          }
          linkb(pmp, bp);
          if(count == 0) {
            goto release;
          }
        }
      }
      else {
        if((bp = allocb(count, BPRI_MED)) == NULL) {
          printf("ioawsvr: count alloc failed\n");
        }
        else {
          while(count--) {
```

```
            *bp->b_wptr++ = *dmp->b_rptr++;
          }
          linkb(pmp, bp);
        release:
#if DEBUG > 1
printf("  Releasing %d blocks\n", getcount);
#endif
          pmp = getq(q);                /* dequeue the M_PCPROTO */
          pmp->b_datap->db_type = M_PROTO;
          while(getcount--) {
            freemsg(getq(q));           /* dequeue the empty M_DATA's */
          }
          putnext(q, pmp);              /* send the WRITE_BUF downstream */
          return;
        }
      }
    }
  }
}
```

7.9 CLOSE ROUTINE

```
static int
ioaclose(q, flag)
queue_t *q;                     /* pointer to read QUEUE */
int flag;                       /* file open flags, 0 for modules */
{

#if DEBUG > 0
printf("ioa close\n");
#endif
  return(0);
}
```

7.10 PRINT MESSAGE DEBUG ROUTINE

```
#if DEBUG > 0
static char *function_str[] = {
    "NO_FUNCTION",      /*  0   no function specified          */
    "IO_CELL",          /*  1   IO command cell                */
    "READ_INST",        /*  2   read instruction               */
    "READ_INST_Y",      /*  3   read instruction y field       */
    "JUMP_INST",        /*  4   jump to new instruction        */
    "JUMP_STAT",        /*  5   jump if status bit set         */
    "READ_REG",         /*  6   read control register          */
    "WRITE_REG",        /*  7   write control register         */
```

```
    "REG_TO_MEM",       /*  8   copy control register to memory */
    "MEM_TO_REG",       /*  9   copy memory to control register */
    "READ_START",       /* 10   read BCW and BAP                */
    "READ_BUF",         /* 11   read memory user data buffer    */
    "WRITE_START",      /* 12   write BCW and BAP               */
    "WRITE_BUF",        /* 13   write memory user data buffer   */
    "EF_START",         /* 14   read BCW and BAP                */
    "EF_BUF",           /* 15   read memory user data buffer    */
    "WRITE_WORD",       /* 16   atomic read-modify-write        */
    "INTERRUPT",        /* 17   generate UYK-44 interrupt       */
    "STAT_TO_MEM",      /* 18   copy status register to memory  */
    "WRITE_STAT",       /* 19   write status register           */
    "TEST_WORD",        /* 20   test memory word                */
    "TEST_AND_SET",     /* 21   test and set memory word        */
    "CHAN_CONTROL",     /* 22   channel control                 */
    "WRITE_EI_WORD"     /* 23   write external interrupt word   */
};

static int
prmsg(mp, s)
mblk_t *mp;
char *s;
{
  struct msg_cntl_part *p;
  mblk_t *bp;
  unsigned short t;

  if(s) {
    printf("%s:\n", s);
  }

  for(bp = mp; bp != NULL; bp = bp->b_cont) {
    switch(bp->b_datap->db_type) {
      case M_PROTO:
      case M_PCPROTO: {
        p = (struct msg_cntl_part *)(bp->b_datap->db_base);
        t = p->function;
        if(t > 0 && t < (sizeof(function_str) / sizeof(*function_str))) {
          printf("  %s", function_str[t]);
        }
        else {
          printf("  %x", t);
        }
        printf(", ack %s", (p->ack == DISCARD) ? "DISCARD" : "REPLY");
        printf(", device %x", p->device);
        printf(", chain %s\n", (p->chain == OUTPUT) ? "OUTPUT" : "INPUT");
        printf("    address %x, data %x, control %x, status %x\n",
         p->address, p->data, p->control, p->status);
```

```
            break;
          }
          case M_DATA: {
            printf("    block of %d bytes\n", bp->b_wptr - bp->b_rptr);
            break;
          }
        }
      }
    }
#endif
```

BIBLIOGRAPHY

Egan, J. I. and T. J. Teixeira. 1992. *Writing A Unix Device Driver: Second Edition*. John Wiley & Sons, New York.

Leffler, S. J., M. K. McKusick, M. J. Karels, and J. S. Quarterman. 1989. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison–Wesley, New York.

Pajari, G. 1992. *Writing Unix Device Drivers*. Addison–Wesley, New York.

Sun Microsystems. 1990. *Writing Device Drivers*. Sun Microsystems, Inc. 2550 Garcia Ave., Mountain View, CA 94043. Part No. 800–3851–10, Rev. A.

Sun Microsystems. *Introduction to Writing Device Drivers: Student Guide. 1*. Sun Microsystems, Inc. 2550 Garcia Ave., Mountain View, CA 94043. Rev. L.1.

Unisys. 1990. *Instruction Set Architecture (ISA) Specification for the AN/UYK–44(V) Computer Set*. Document no. SEA–ISA–4412.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1 AGENCY USE ONLY *(Leave blank)*	2 REPORT DATE	3 REPORT TYPE AND DATES COVERED
	May 1993	Final; May 1993

4 TITLE AND SUBTITLE
UNIX STREAMS EMULATION OF AN INPUT/OUTPUT CONTROLLER (IOC) FOR AN EMBEDDED AN/UYK-44(V) PROCESSOR

5 FUNDING NUMBERS
AN: DN587574
PE: OMN 0305167G
PN: CC3001

6 AUTHOR(S)
D. R. Wilcox and P. N. Pham

7 PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Command, Control and Ocean Surveillance Center (NCCOSC)
RDT&E Division
San Diego, CA 92152-5001

8 PERFORMING ORGANIZATION REPORT NUMBER
TD 2533

9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Naval Space and Warfare Systems Command
Code 398277
Washington D.C. 20360-5100

10 SPONSORING/MONITORING AGENCY REPORT NUMBER

11 SUPPLEMENTARY NOTES

12a DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b DISTRIBUTION CODE

13 ABSTRACT *(Maximum 200 words)*

This report presents the implementation of a software interface between a Unix operating system executing on a Navy DTC-2 VMEbus computer workstation and a VMEbus AN/UYK-44(V) enhanced processor (EP) embedded within that workstation. The interface software employs the Unix STREAMS mechanism to emulate the AN/UYK-44(V) input/output controller (IOC) function. Emulated capabilities include IOC page register groups, multiple input/output controllers, input, output, external function, and forced external function transfers, and external interrupt. The AN/UYK-44(V) application directs the interface by supplying AN/UYK-44(V) command instructions and chain programs in memory. The Unix application communicates with the interface using the Unix file system calls.

14 SUBJECT TERMS
command control
computer sciences
AN/UYK-20/43144
software tools
support software

15 NUMBER OF PAGES
117

16 PRICE CODE

17 SECURITY CLASSIFICATION OF REPORT	18 SECURITY CLASSIFICATION OF THIS PAGE	19 SECURITY CLASSIFICATION OF ABSTRACT	20 LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT

21a NAME OF RESPONSIBLE INDIVIDUAL	21b TELEPHONE (*include Area Code*)	21c OFFICE SYMBOL
D. R. Wilcox	(619) 553-5467	Code 412

INITIAL DISTRIBUTION

Code 0012	Patent Counsel	(1)
Code 40	R. C. Kolb	(1)
Code 41	A. G. Justice	(1)
Code 4102	M. L. Crowley	(1)
Code 4102	H. Gold	(1)
Code 4102	G. B. Myres, Jr.	(1)
Code 4103	L. J. Core	(1)
Code 4103	D. F. Romig	(1)
Code 412	J. L. Conrath	(1)
Code 412	D. K. Fisher	(1)
Code 412	P. N. Pham	(1)
Code 412	L. A. Rasmus	(1)
Code 412	D. G. Sheriff	(1)
Code 412	V. D. Tran	(1)
Code 412	D. R. Wilcox	(20)
Code 413	R. E. Johnston	(1)
Code 961	Archive/Stock	(6)
Code 964B	Library	(2)

Defense Technical Information Center
Alexandria, VA 22304–6145 (4)

Center for Naval Analyses
Alexandria, VA 22302–0268

NCCOSC Washington Liaison Office
Washington, DC 20363–5100

Navy Acquisition, Research and Development
Information Center (NARDIC)
Washington, DC 20360–5000

GIDEP Operations Center
Corona, CA 91718–8000

NCCOSC Division Detachment
Warminster, PA 18974–5000

Naval Sea Systems Command
Washington, DC 20362–5101 (5)

Naval Surface Warfare Center
Dahlgren, VA 22448–5000 (2)

MCTSSA
Camp Pendleton, CA 92055–5171 (2)

Carnegie Mellon University
Pittsburgh, PA 15213–3890

Paramax
St. Paul, MN 55164–0525 (5)